MASTER EN SCIENCES INFORMATIQUES

Sur base d'une double spécification - Réseaux de Pétri/L.O.T.O.S. - de Fip

Hublet, Corine

Award date:
1990

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix

Institut d'Informatique

"Sur base d'une double spécification - Réseaux de Pétri/ L.O.T.O.S. - de Fip "

Promoteur : Jean FICHEFET

Mémoire présenté par Corine HUBLET
en vue de l'obtention du titre de
licenciée et maître en
informatique.

Année académique 1989-1990.

Avant-propos

Les travaux présentés dans ce mémoire ont été effectués au Laboratoire d'Automatique et d'Analyse des Systèmes du Centre National de la Recherche Scientifique à Toulouse, en France. Nous tenons à témoigner notre gratitude à Monsieur le Professeur Jean FICHEFET, notre promoteur, qui a permis l'obtention de ce stage et qui nous a aidée lors de la réalisation de cet ouvrage.

Nous remercions tout particulièrement Monsieur P. AZEMA, notre maître de stage, membre du groupe "Outils et Logiciels pour la Communication", qui nous a accueillie dans ce groupe et a dirigé nos travaux.

Nous exprimons également notre reconnaissance à Monsieur Khalil DRIRA et à tous les autres membres du groupe OLC qui, par leur soutien et leurs remarques constructives, ont permis d'améliorer les travaux présentés. Que ne soient pas oubliés ici, Ph. MOISSE, A. MOISSE et A. SERVAIS pour leur aide amicale.

Résumé

Lors de la spécification de systèmes distribués ou complexes, tel que le protocole de communication Fip, le langage naturel s'avère souvent inadéquat pour élaborer une spécification qui soit claire, complète et précise. La seule solution pratique consiste, dès lors, à recourir à un formalisme reposant sur des bases mathématiques solides, comme par exemple les réseaux de Pétri ou L.O.T.O.S. A titre d'exemple, nous avons spécifié Fip en L.O.T.O.S. sur base, notamment, du modèle des réseaux de Pétri élaboré par Nordgard.

Abstract

When at the time of the specification of distributed or complex systems such as Fip communication protocol, the natural language often proves inadequate to build up a clear, complete and accurate specification. The only practical solution consists then in resorting to a formalism based on strong mathematical bases as for instance the Petri Net or L.O.T.O.S. As illustration, we specified Fip in L.O.T.O.S. mainly based on Petri Networks worked out by Nordgard.

Introduction générale

Quel que soit leur domaine d'activité, les systèmes distribués occupent, de nos jours, une importance fondamentale. Pensons, par exemple, au réseau bancaire de guichets automatiques ou encore au réseau local présent aux facultés. Ils connaissent une prolifération accrue dans le monde industriel car ils permettent l'obtention d'informations disparates nécessaires à la production. De tels systèmes offrent de multiples avantages dont une amélioration certaine des performances, due au parallélisme.

Cet aspect de répartition des informations implique de multiples échanges de données qui doivent être définis de manière complète et non ambigüe par des règles de dialogue ou protocoles tel que Fip. Ces protocoles doivent être testés avant même d'être implantés afin d'éliminer tout risque inutile. Seule une approche reposant sur des bases mathématiques solides permet de satisfaire à cet objectif ; citons par exemple, l'approche adoptée par les réseaux de Pétri. Cependant, en appliquant ces réseaux à différents protocoles, il est vite devenu évident que leur pouvoir d'expression était insuffisant. Ce pouvoir ne peut spécifier, à un niveau de détail adéquat, des protocoles complexes tel que ceux du modèle de référence pour l'interconnexion des systèmes ouverts (OSI) de l'ISO ("International Standard Organization") . Cette limite s'est peu à peu restreinte par l'apparition de langages qui s'appuient sur des extensions des réseaux de Pétri. Citons, à titre d'exemple, le langage P.I.P.N. développé et utilisé, quasi exclusivement, au Laboratoire d'Automatique et d'Analyse des Systèmes du CNRS à Toulouse. L'ISO a lui même mis sur pied un groupe chargé de développer deux langages formels : Estelle et L.O.T.O.S. Ces langages nécessitent, dans l'optique d'une acceptation et d'une utilisation internationales, des environnements logiciels adéquats et ont suscité le projet S.E.D.O.S.

("Software Environment for Distributed Systems"). Ce projet avait notamment comme finalité de prouver l'adéquation de ces langages à la spécification de protocoles grâce à des spécifications de protocoles existants.

Sur base du modèle des réseaux de Pétri élaboré par Nordgard, le protocole Fip a fait l'objet d'une spécification et d'une analyse à l'aide du langage P.I.P.N. Le langage L.O.T.O.S. ayant été récemment normalisé et le protocole étant lui-même en voie de normalisation, le L.A.A.S. a souhaité le spécifier en L.O.T.O.S et examiner les liens entre ces trois formalismes de spécification : les réseaux de Pétri, P.I.P.N. et L.O.T.O.S. Ces deux travaux constituent le cadre de notre mémoire : nos spécifications de Fip en L.O.T.O.S. se trouvent en annexes tandis que notre étude comparative est présentée au chapitre quatre.

En conséquence, nous consacrons le premier chapitre à un bref exposé du protocole de communication Fip et plus particulièrement de la couche liaison de données sur laquelle s'attelle notre travail.

Le deuxième chapitre est consacré à un rappel du modèle des réseaux de Pétri et de ses dérivés ; il nous permet d'introduire les caractéristiques essentielles du langage P.I.P.N.

Le troisième chapitre présente le langage normalisé L.O.T.O.S. dont la sémantique repose sur les langages ACT-ONE et C.C.S. ("Calculus on Communicating Systems"). Une version simplifiée du langage C.C.S. est présentée et fait le lien avec les systèmes de transitions, ancêtres des réseaux de Pétri. Elle met en évidence le dynamisme d'une telle description et le manque de types de données. Le langage ACT-ONE permet ensuite d'exposer les limites d'un langage

uniquement fondé sur les types abstraits de données et l'utilité d'un langage mixte tel que L.O.T.O.S. Ce dernier est décrit sur base d'une spécification de la couche liaison de données de Fip dont les procédés d'élaboration sont également expliqués. Enfin, un aperçu des objectifs du projet S.E.D.O.S. est donné en vue d'en souligner la contribution.

Le quatrième chapitre décrit, sur base des observations faites lors de l'étude des trois spécifications finalement disponibles de la couche liaison de données de Fip, les différences fondamentales qui distinguent les réseaux de Pétri, P.I..P.N. et L.O.T.O.S. Cette analyse nous conduit à ébaucher quelques conseils d'utilisation de ces différents formalismes, analyse dont nous soulignons ensuite la portée restreinte.

Chapitre 1 : Le protocole Fip
(Flux d'Information de Processus)

I. Présentation

I.1. Introduction

Fip est un modèle de spécification de bus de terrain. Les bus de terrain sont des petits réseaux locaux spécialisés dans l'acquisition et le routage de données et utilisés par les entreprises ayant besoin de systèmes intégrés de production. Les machines connectées à ces bus ont des besoins en information très différents mais prévisibles. Souvent ces systèmes intégrés accompagnent le bus de terrain de bus de coordination et de gestion technique apportant les fonctionnalités nécessaires à une bonne sûreté de fonctionnement.

Le type de bus qu'implémente Fip est appelé bus à diffusion, c'est-à-dire qui permet à toute machine connectée de percevoir toute information émise. Ainsi, Fip offre un réseau de terrain ouvert, exploitable en temps réel et descendant jusqu'au niveau même des capteurs et actionneurs. Mais, "bien que Fip soit une norme homologuée, il connaît encore des problèmes de définition de ses différentes couches" [Le-Th 89].

I.2. Fonctions de base d'un bus de terrain

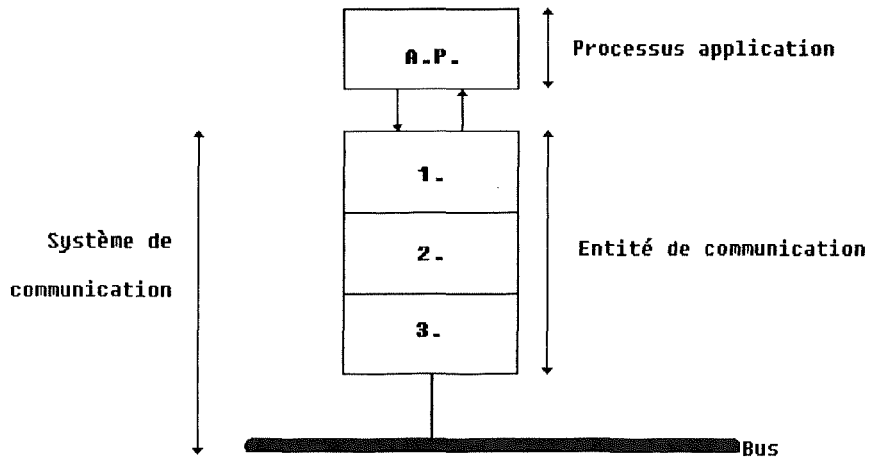
Les exigences des échanges d'entrées/sorties industrielles sont telles que le bus se doit d'être transparent et de n'imposer aucune contrainte structurelle. Les entrées/sorties concernées peuvent être de plusieurs types : commande de dispositifs, demande de leurs états et entrées/sorties de valeurs, ces dernières se faisant souvent à intervalle fixe. Dans cette optique, tout dispositif sera considéré comme un

objet à quatre éléments : commande, état, entrée et sortie. Les opérations possibles devant bien entendu être adressables, seront l'écriture d'une commande, la lecture d'états, l'entrée ou la sortie de valeurs.

Pour un tel bus, la structure idéale sera décentralisée car elle offrira la possibilité d'améliorer certains aspects tels que les coûts et la modularité [Le-Th 89] ; de plus ses services couvriront, pour faire un parallèle avec le modèle OSI de l'ISO, les couches un à sept.

I.3. Présentation technique des services du système de communication Fip

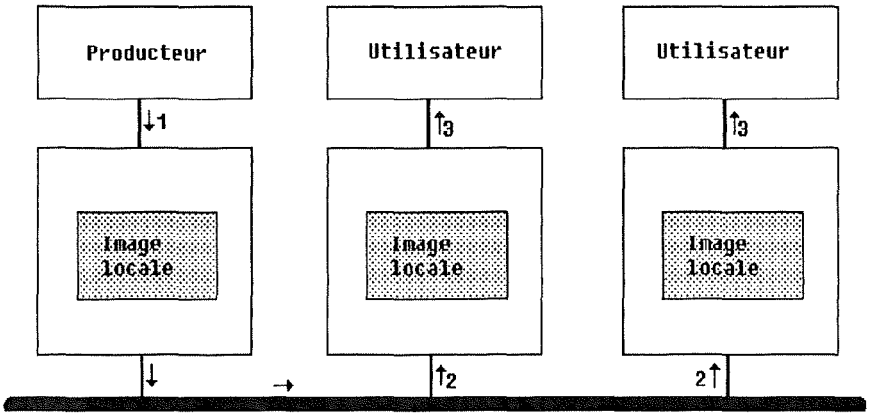
Le système de communication Fip permet aux différents processus impliqués dans une application répartie d'échanger des informations via un ensemble de services et ressources, appelé "entité de communication" et associé à chacun d'eux. Chaque entité de communication est architecturée en trois couches selon une hiérarchie proche du modèle OSI à sept couches.



I.3.1. La couche application

L'idée de base sur laquelle repose l'implémentation des services de lecture et d'écriture de valeurs de variables, services offerts au processus d'application utilisateur, est la mise en relation du producteur de la variable concernée et de ses utilisateurs. Une information ou variable ne possède en effet qu'un seul producteur.

Chaque entité de communication possède une image locale des variables du système de production dont les valeurs sont mises à jour par une fonction de transfert. Ces valeurs peuvent cependant diverger d'une entité à l'autre en raison d'erreurs de transmission. Des mécanismes assureront l'intégrité temporelle et spatiale des échanges [Le-Th 89].



I.3.2. La couche liaison de données

Afin d'implanter les services de la couche application, Fip propose des services quelque peu différents de ceux traditionnellement offerts. Le modèle général des communications usuelles, ou modèle bipoint, est ainsi écarté au profit d'un modèle s'appuyant sur le principe de la

diffusion générale. Dans cette optique, le principe d'adresse est l'adressage "source" où seul l'émetteur est désigné. C'est à ce niveau qu'intervient l'une des originalités de Fip : l'adressage représente le nom d'un objet parfaitement identifié de l'application. En effet, l'objet d'un bus de terrain étant, par exemple, la transmission des valeurs issues des capteurs à des actionneurs, l'information à transmettre est identifiée et répertoriée sous forme d'une nomenclature ordonnée d'objets. Chaque objet est désigné par un nom unique et global pour l'application. Ce mécanisme de désignation permet de considérer Fip comme un système de mise à jour et de gestion de base de données réparties [Le-Th 89].

La nomenclature ordonnée d'objets se répète de manière cyclique et confère à Fip la fonction principale de scrutation périodique. Un autre trafic apériodique assure les échanges sur demandes, tels la retransmission d'une valeur. En effet, aucun mécanisme d'acquittement n'a été mis en place de par le principe de diffusion et la sécurité de transmission de données n'est assurée que dans une certaine mesure.

Cette couche fera l'objet d'une étude plus approfondie dans la suite du travail. En effet, notre objectif est d'en spécifier le trafic périodique en L.O.T.O.S. (annexe C), sur base du modèle de réseaux de Pétri élaboré par Nordgard (annexe A) et sur base d'un modèle basé sur les réseaux prédicat/transition (annexe B). Ces modèles seront eux aussi expliqués par la suite.

I.3.3. La couche physique

"La couche physique s'occupe principalement de la connectique et de la définition des mécanismes de redondance de la voie de transmission" [Le-Th 89].

II.2. Deux types de trafics associés à la couche liaison de données

II.2.1.Le trafic périodique

- Les services offerts à la couche application

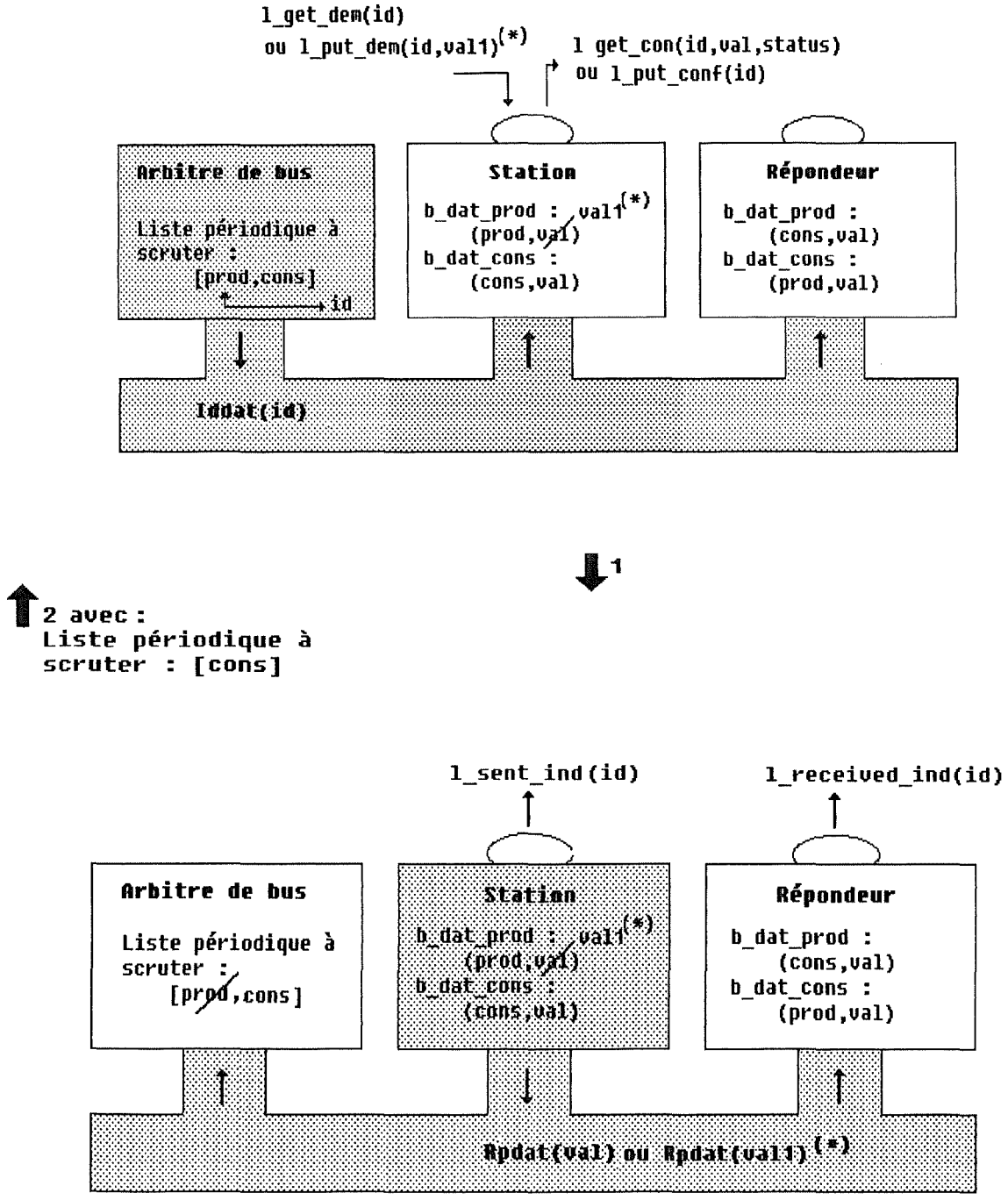
La couche liaison de données offre à la couche application les services de lecture, d'écriture et de transfert de fichiers.

Le service de lecture permet l'extraction de l'information associée à un identificateur, ou valeur associée à un objet parfaitement identifié de l'application, via la primitive "l_get_dem(identificateur)". La confirmation relative au bon déroulement de cette primitive sera fournie par un statut via la requête : "l_get_conf(identificateur, statut, valeur)" .

Le service d'écriture permet à la couche application, si son processus d'application utilisateur est le producteur de l'identificateur, de lui associer une autre valeur via la primitive : "l_put_dem(identificateur, nouvelle valeur)". La primitive "l_put_conf(identificateur, statut)" confirmera ou non le bon déroulement de l'opération précédente.

Enfin, le service de transfert de fichier ou de buffer permet l'émission (ou la réception) de la valeur associée à un identificateur sur le médium, et un compte rendu de ce service vers la couche application au travers de la primitive "l_sent_indic(identificateur)" ou respectivement "l_received_indic(identificateur)"

- Spécification graphique du service de scrutation cyclique (ou trafic périodique)



II.2.2. Le trafic apériodique

- Les services offerts à la couche application

La couche application peut effectuer deux types de demande d'échanges apériodiques, spécifiés ou libres, qui se distinguent dans la manière de les adresser. Nous n'aborderons dans le cadre de ce mémoire, que les échanges apériodiques spécifiés dont la station prend l'initiative.

Les diverses primitives de services à l'interface application sont :

. "l_update_spec.dem (identificateur, suite) " par laquelle la couche application demande sur le bus les identificateurs de la liste "suite". Cette demande est véhiculée sur l'identificateur évoqué

. "l_update_spec.conf (identificateur, statut)" qui fait suite à une demande de service apériodique spécifié, et qui en signale le déroulement via le "statut".

- Spécification en langage naturel du service apériodique spécifié

. Lors de la réception d'une demande apériodique spécifiée, la couche liaison de données stocke la suite des identificateurs de la suite à scruter, dans le buffer réservé à l'identificateur sur lequel a été véhiculé la demande (" b_req(identificateur,suite,rq) "). Ce buffer est vide a priori, ce que nous indique son champ "rq" de valeur nulle. La mise à jour de la suite à scruter change la valeur de ce champ à un. Cependant, il se peut que le buffer soit déjà rempli par une suite laquelle doit être envoyée à l'arbitre

("rq" est déjà égal à un). Dans ce cas, l'ancienne liste est écrasée par la nouvelle et la primitive "l_update_spec.conf (identificateur, "écrase") " est envoyée à la couche application. Le champ "rq" ne subit aucune modification.

. Lors du trafic périodique, si la station reçoit la requête d'émission d'une valeur (" iddat(identificateur) ") dont elle est productrice, elle vérifie le champ "rq" du buffer associé à cet identificateur. Si ce champ équivaut à la valeur nulle, aucune demande d'échanges apériodiques spécifiés n'a été effectuée. La station émet alors la valeur associée à cet identificateur sur le bus (via la primitive "rpdat(valeur) "). Si par contre, ce champ a une valeur de un, la station prévient l'arbitre qu'elle désire une circulation apériodique au moyen de la primitive "rpdat_rq (valeur) ").

. Une fois le trafic périodique terminé, l'arbitre aborde la fenêtre apériodique et émet une primitive "id_rq(identificateur) " où l'identificateur est le premier d'une liste 'fifo' sur lequel a été effectuée une demande apériodique.

. La station ayant émis la demande sur cet identificateur, fournit à l'arbitre la suite d'identificateurs dont elle désire la retransmission à l'aide de la primitive "rp_rq (suite) " et indique à sa couche application que sa demande a été prise en compte par l'arbitre (via "l_update_spec.conf (identificateur,"ok") "). Celui-ci scrute la suite des identificateurs. Pour chacun, il envoit la primitive "iddat(identificateur) " et attend en retour la valeur qui lui est associée via la primitive "rpdat(valeur) ". Lorsque sa liste fifo est vide, l'arbitre aborde la fenêtre messagerie (que nous ne verrons pas ici).

III. Conclusion

La description de Fip en langage naturel nous a permis de mettre en évidence certaines de ses caractéristiques, tel l'adressage source. Une telle description sert généralement de référence pour la coopération parmi les nombreux concepteurs d'un protocole. Cependant, par son manque de sémantique formelle, cette technique de description s'avère inadéquate dans de nombreux cas car elle aboutit à de multiples erreurs ou à des comportements tant inattendus qu'indésirables [Bo-Su 80]. Afin de pallier ces inconvénients, des techniques plus rigoureuses ont vu le jour sous l'appellation de "techniques de description formelles ". Une telle technique est une méthode de spécification, basée sur un langage de description, lui-même doté d'une syntaxe et d'une sémantique formelle [ISO/IEC 89], tels que les réseaux de Pétri, les types de données abstraits algébriques, l'approche algébrique C.C.S. ou L.O.T.O.S. ...

Dans un premier temps, nous aborderons les réseaux de Pétri et le langage de prototypage rapide "P.I.P.N." qui nous ont servi de référence pour l'élaboration de la spécification L.O.T.O.S.. Ensuite, dans le chapitre 3, nous développerons le langage de spécification qu'est L.O.T.O.S. et ses deux théories sous-jacentes, à savoir l'approche algébrique C.C.S. et les types abstraits.

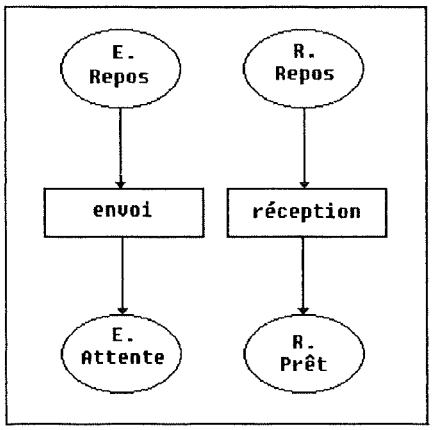
Chapitre 2 : Quelques formalismes supports pour la spécification de protocoles

I. Les réseaux de Pétri

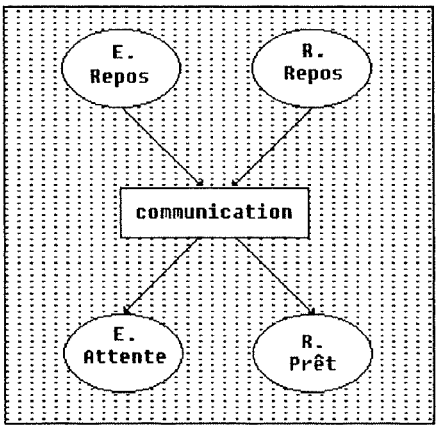
I.1. Introduction informelle

Les réseaux de Pétri ont été introduits au début des années soixante par C.A. Pétri. Ils permettent, en particulier, de modéliser et d'analyser des systèmes de processus concurrents et parallèles. Pour ce faire, ils se basent sur l'approche orientée "état" : le comportement dynamique du système est modélisé sur base des différents états qui le composent et qui résultent de la survenance d'événements ou actions. Par conséquent, un réseau de Pétri peut être vu comme un graphe orienté dont les noeuds sont les états et les actions représentés respectivement par des cercles et des traits. Les premiers sont désignés par la notion de place et les seconds par celle de transitions. Un arc relie une place donnée à une transition ou inversément. En outre, les réseaux de Pétri peuvent également être utilisés, malgré leur médiocre pouvoir de description, dans une optique de spécification.

Il apparaît dès lors que l'une des caractéristiques fondamentales est l'accent mis sur la structure interne du système et la combinatoire des états. Un tel formalisme s'avère particulièrement utile lors de l'élaboration de systèmes communiquant par l'intermédiaire de variables partagées. Citons pour mémoire la mutuelle exclusion. Cependant, il n'est plus nécessaire, comme c'était le cas pour les systèmes de transitions, ancêtres des réseaux de Pétri, d'expliciter tous les changements d'état : il suffit désormais de les caractériser par une procédure. Supposons, par exemple, qu'un émetteur envoie un message à un récepteur. Celui-ci accusera réception par l'envoi d'un message ; dans ce cas, la modélisation se fera comme suit :



Système de transitions



Réseau de Pétri

De par leur formalisme tant graphique que formel, les réseaux de Pétri permettent d'exprimer le parallélisme, la concurrence ou encore la synchronisation et ce, de manière non ambiguë.

I.2. Définition

Un réseau de pétri est un quadruplet <P,T,I,O> où :

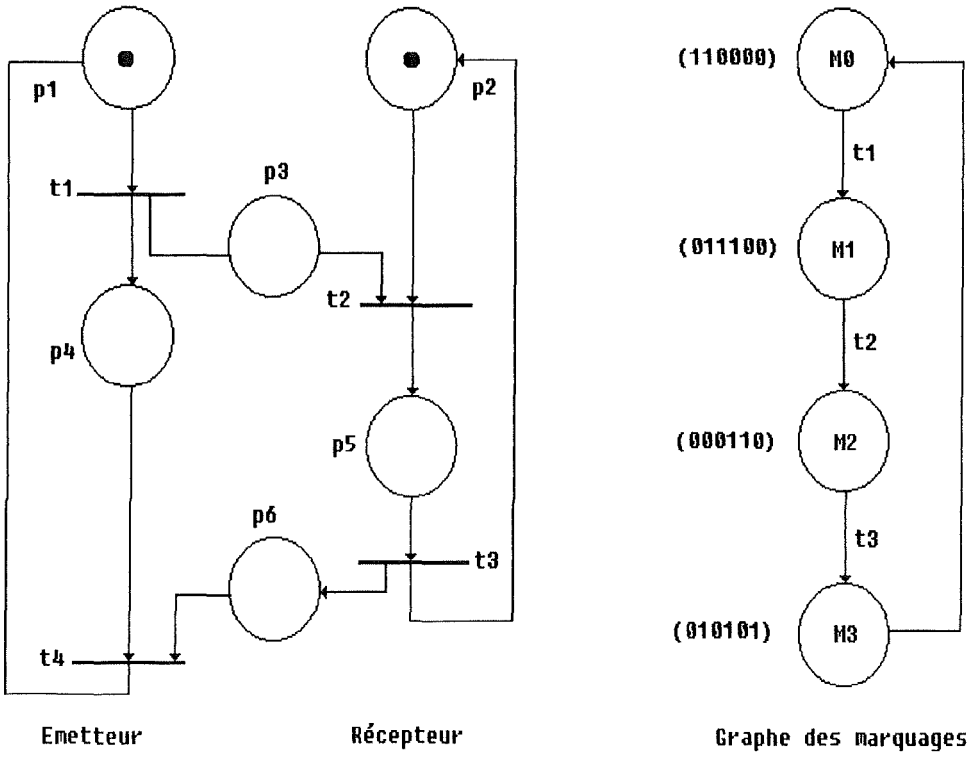
- P est un ensemble fini de places (ou états),
- T est un ensemble fini de transitions (ou actions),
- I/O est l'ensemble des fonctions en entrée/sortie qui reprennent les conditions devant être vérifiées en entrée/sortie de chaque action.

L'ensemble des places et transitions décrivent l'aspect statique du système. L'état de celui-ci est alors modélisé à l'aide d'un marquage que l'on fait évoluer en exécutant les actions, c'est-à-dire en franchissant les transitions. Ce marquage s'interprète comme la présence de ressources d'un certain type. Le franchissement d'une transition représente une action qui a lieu lorsqu'un nombre de ressources est présent, c'est-à-dire lorsque les conditions en entrées ou

"préconditions" sont satisfaites. L'action emploie ces ressources pour en produire d'autres qui corresponderont aux conditions en sortie ou "postconditions". Dès lors, à partir du marquage de départ ou marquage initial, un diagramme d'états du comportement dynamique du réseau peut être élaboré. Il reprend l'ensemble des séquences de transitions franchissables et est couramment désigné sous le terme de "graphe des marquages".

Exemple

Le réseau ci-dessous représente la communication entre un émetteur et un récepteur selon un processus : question/réponse.



où (110000) est équivalent au nombre de marques présentes dans (p1 p2 p3 p4 p5 p6).

I.3. Principales propriétés

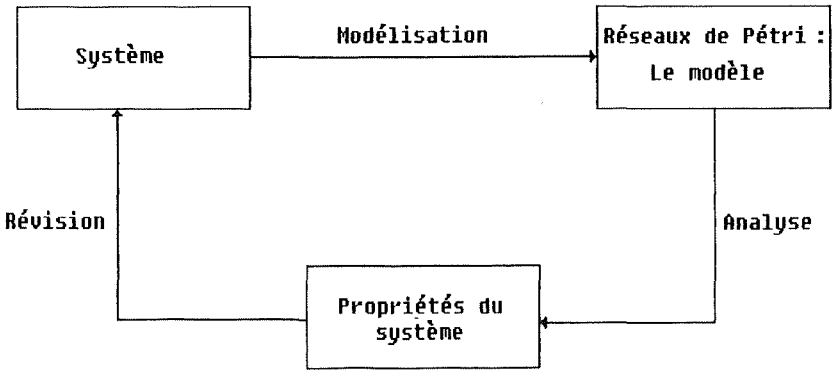
Elles peuvent être classées en deux catégories : les propriétés dynamiques et les propriétés structurelles.

- Les propriétés dynamiques ou générales. Elle doivent, à priori, être satisfaites quelque soit le réseau. Les propriétés attendues sont généralement que le réseau soit borné, vivant ou encore réinitialisable. La première propriété implique un nombre fini d'états ce qui permet l'implantation du système modélisé. La deuxième propriété, ou vivacité, démontre qu'il n'y pas d'interblocage ou d'interactions non exécutables. Enfin, la dernière propriété met en évidence le fonctionnement cyclique du système. Ces propriétés dépendent du marquage initial et sont liées à l'évolution du réseau. Leur vérification se fait généralement par la construction du graphe des marquages.

- Les propriétés structurelles ou spécifiques. "Elles permettent de prouver des assertions sur le marquage ou sur les séquences de tir. Elles dépendent de la sémantique associée aux éléments du réseau (place et transition) et non plus du marquage initial" [Courtiat 87]. Leur analyse repose sur la théorie de l'algèbre linéaire, laquelle peut également être utilisée lorsque la construction du graphe des marquages s'avère lourde ou impossible. Nous renvoyons le lecteur intéressé à ce sujet à [Peterson 81] pour de plus amples informations.

I.4. Utilité

Si les réseaux de Pétri permettent la modélisation et la spécification des systèmes distribués, il n'en reste pas moins qu'en pratique leur intérêt ne réside que dans leur puissance de modélisation. Celle-ci offre de multiples avantages : d'une part, elle permet une modélisation hiérarchique du système étudié. Une place peut être remplacée par un sous-réseau en vue d'une représentation plus fine (ou inversément). Cependant, lorsque la taille du système modélisé s'accroît, elle devient vite prohibitive rendant nulle toute compréhension ou utilisation. Il est dès lors regrettable que ces réseaux n'offrent aucun moyen naturel de construire un modèle à partir de composants élémentaires. D'autre part, les propriétés d'un protocole peuvent être analysées par l'utilisation des résultats classiques sur la validation des réseaux de Pétri : vérification des propriétés dynamiques ou structurelles du système étudié. Ainsi, il est possible de déterminer si le protocole est réinitialisable, c'est-à-dire s'il existe des états à partir desquels il n'est pas possible de revenir à l'état initial; ou encore s'il est vivant : s'il existe des états à partir desquels une transition donnée n'est pas accessible; ou encore, de montrer que le protocole progresse effectivement. Dès lors, les réseaux de Pétri sont souvent considérés comme un outil d'analyse auxiliaire. Le système est spécifié à l'aide d'une technique de conception conventionnelle et fait ensuite l'objet d'une modélisation et d'une analyse à l'aide des réseaux de Pétri. En L.O.T.O.S. , le graphe des marquages peut-être déduit systématiquement du système spécifié, grâce aux travaux effectués par Garavel.



I.5. Les réseaux prédicats/transitions : une extension aux réseaux de Pétri

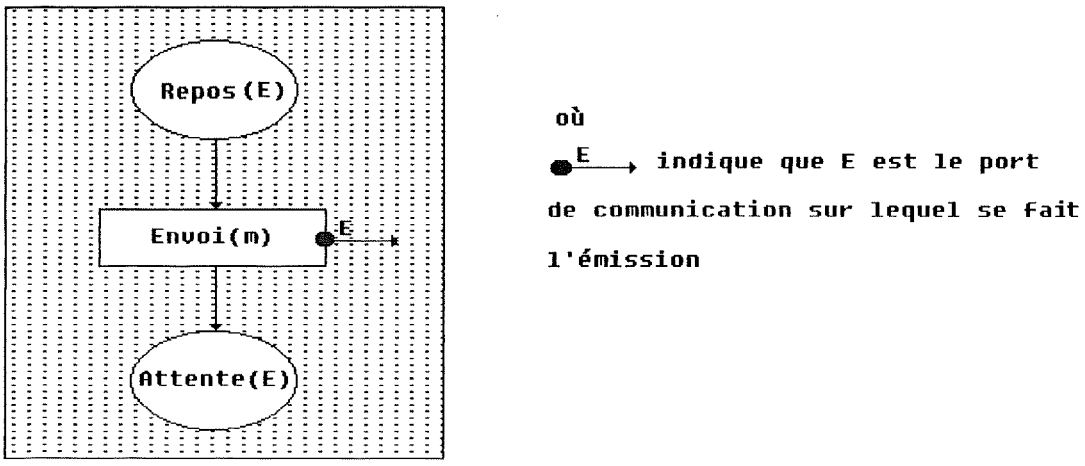
Les réseaux prédicats/transitions sont des réseaux de haut niveau construits sur la logique du premier ordre : à chaque place ou transition est associé, non plus une proposition, mais un prédicat. Ce qui permet, d'une part une ou plusieurs représentations possibles d'un même réseau et d'autre part la prise en considération de données. De cette façon, nous n'aurons plus deux propositions : E. Repos, R. repos, lors de l'émission d'un message, mais repos(X) où X est une variable qui pourra être instanciée à E ou à R. Ce qui, si nous reprenons l'exemple, sera noté :

tr(communication, [Repos(E),Repos(R)], [Attente(E), Prêt(R)])

De plus, les réseaux prédicat/transition nous permettent d'associer au réseau une interprétation, par exemple, en définissant l'univers de discour. Ainsi, dans la réalité, l'émetteur ne doit pouvoir envoyer un message que lorsqu'il a établi la liaison avec son correspondant.

tr(communication, [Repos(E),Repos(R)], [Attente(E), Prêt(R)], SI connecte(E,R))

Cependant, si une telle modélisation, voire spécification, constitue une abstraction du système envisagé, elle n'offre toujours pas la compositionnalité. Afin d'y remédier, le concept de "label" ou d'étiquette a été introduit : il permet de décrire les actions atomiques de communication entre le système et son environnement. Cette étiquette comporte non seulement le message proprement dit, mais aussi sa direction (réception ou émission) ainsi que le port de communication sur lequel il est émis.



Ce qui se notera :

tr(envoi(m),E ! m, [Repos(E)], [Attente(E)])

où E ! m (? m) signifie l'émission de m (la réception de m) sur le port E.

Mais de ce fait, une partie de la dynamique du système nous échappe, même si le comportement de ces constituants induit le comportement global du système. En effet, nous ne percevons plus que les comportements locaux : celui de l'émetteur ou du récepteur.

Cette dernière extension servira de référence sémantique au langage de prototypage rapide P.I.P.N. avec lequel nous avons travaillé au L.A.A.S.

II. Le langage de prototypage rapide : P.I.P.N.

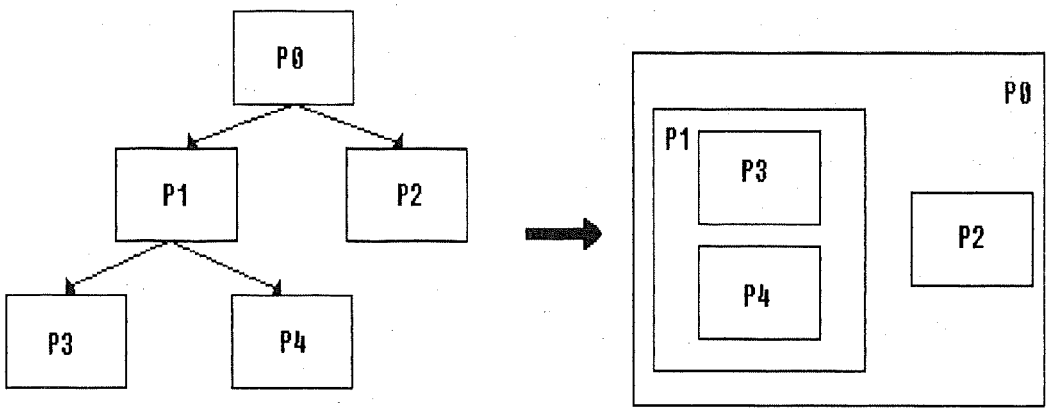
II.1. Introduction

P.I.P.N. est un langage et un outil de prototypage rapide de système parallèle, qui emprunte tantôt à la théorie des réseaux prédicats/transitions labellés, tantôt à un autre formalisme de modélisation de systèmes communicants et parallèles : C.C.S. ("Calculus on Communicating Systems"). Ce dernier constitue également l'une des deux théories sur laquelle se base L.O.T.O.S. et que nous aborderons dans le chapitre suivant. Cependant, soulignons d'ores et déjà, que C.C.S. permet de décrire le comportement d'un système de manière structurée grâce à l'utilisation d'opérateurs. La sémantique de certains de ces opérateurs, dont celui de composition parallèle, a été reprise dans le langage P.I.P.N. en vue d'offrir la compositionnalité.

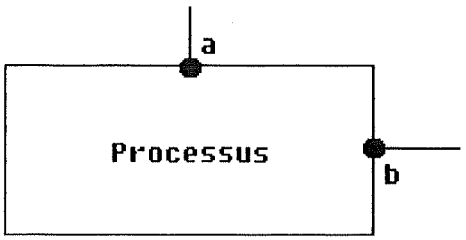
II.2. Modèle

Dans le langage P.I.P.N., tout système est décrit par un modèle. Un modèle se structure en une hiérarchie père-fils de

modules communicants qui peut être représentée soit par un arbre de module, soit par leur encapsulation.



Un module représente un réseau prédicats/transitions étiqueté et peut être envisagé comme une boîte noire, un processus qui interagit avec son environnement via ses canaux de communication ou point d'accès encore appelé point d'interaction.



où a,b représentent les ports de communication

Ces points d'interactions déterminent le sens de la communication. Aussi, nous distinguons les points d'interactions en entrée, notés 'input(a)', de ceux en sortie ('output(b) ') où a et b désignent le nom des ports de communication. Cette notation nous permet également de décrire les entrées et sorties sur un même port et de vérifier

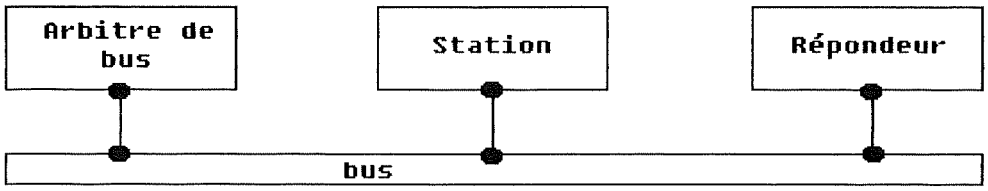
que la communication entre deux modules frères s'effectue sur des ports complémentaires ('input/output'). Ces ports sont interconnectés par une clause 'connect' :

```
connect(station(output(phy(_X))),medium(input(phys(_X)))).
```

où . '_X' désigne une variable
.'station(output(phy(_X)))' désigne le port en sortie de la station acceptant les messages qui instancie '_X'
.'medium(input(phys(_X)))' est le port en entrée du médium acceptant les instances de '_X'

Cet exemple illustre le fait que les interactions en sortie de la 'station' correspondent aux interactions en entrée du 'medium'.

L'interconnexion de deux modules sur des ports complémentaires peut s'illustrer de la manière suivante :



Le protocole Fip

Soulignons au passage, que seuls deux modules frères sont habilités à communiquer ensembles [Lloret 88].

II.3. Syntaxe

Un module est une instance d'un réseau prédicats/transitions étiqueté et est décrit à l'aide d'une interface, d'une configuration et d'un corps. Le corps constitue la partie invariante ou génératrice du système. Celle-ci est instanciée dans l'interface et fait éventuellement appel aux éléments de la configuration lors des conditions de tests. Ces éléments décrivent des relations sur les données par des procédures Prolog.

Une telle découpe permet une réemploi du corps dans d'autres modules. Par exemple, lorsque nous avons spécifié, à titre d'exercice, le réseau téléphonique dans ce langage, nos deux appelants étaient des maisons distinctes qui effectuaient la même démarche : l'appel téléphonique. Ils avaient donc même corps. De plus, la syntaxe du langage est exécutable. En effet, chaque élément syntaxique est un fait ou une clause décrite dans le langage Prolog. Le choix de ce langage est lié à ses possibilités. Il permet le calcul des prédicats, le prototypage rapide et un style déclaratif [Az-Ve-Ll 89].

II.4. Utilité

Le langage P.I.P.N. permet, non seulement, une approche de conception descendante et un prototypage rapide, mais également, une vérification, tant modulaire que compositionnelle, qui nous permet de prouver que l'algorithme décrit fournit bien le service pour lequel il a été conçu, et ce notamment, par le graphe des marquages associé aux réseaux prédicat/transition labellés.

III. Conclusion

Bien que nous ayons regroupé les réseaux de Pétri dans les 'formalismes supports pour la spécification de protocoles', ils constituent avant tout un outil de modélisation. En effet, ils nous permettent de représenter tout système de manière formelle dans une optique de validation. Cependant, comme le souligne J.P. Courtiat dans sa thèse, la première approche dans la modélisation d'un protocole consiste en sa spécification. Celle-ci s'applique à un énoncé informel et peut s'élaborer soit dans un langage naturel (français ou anglais) soit dans un langage formel. Elle permet d'expliciter une entité en la nommant et en lui associant une définition intuitive, avant de la représenter et/ou d'en décrire les propriétés pertinentes. Aussi, un langage de spécification doit non seulement faire référence au niveau de sa définition sémantique à un formalisme tel que les réseaux de Pétri mais il doit offrir un ensemble de facilités supplémentaires qui lui permettent d'une part de décrire les types de données manipulés et leur traitement, et d'autre part de structurer la spécification en composants élémentaires. Les réseaux prédicats/transitions et prédicats/transitions labellés nous permettent de passer progressivement à des formalismes de spécification. Ceux-ci offre une vue plus abstraite que les réseaux de Pétri bien qu'ils restent focalisés sur la description des transitions et donc la structure interne du système dont nous pouvons nous abstraire assez aisément par l'utilisation d'un langage tel que L.O.T.O.S. Dans ce langage, la spécification d'un système réparti est fonction de son comportement externe observable. Une telle approche est particulièrement utile lorsqu'on désire spécifier le comportement attendu par l'utilisateur du système. Notons qu'il s'agit désormais d'un langage de spécification normalisé (depuis 1988)

Chapitre 3 : L.O.T.O.S.
"Language of Temporal Ordering Specification"

I. Introduction

L.O.T.O.S. est un langage principalement conçu pour la spécification formelle des protocoles et des services du modèle OSI . Pour ce faire, quelques concepts de base ont été introduits : l'interaction, le processus, les types abstraits de données et l'ordonnancement temporel. En effet, contrairement aux techniques de spécification formelle basées sur la représentation des états du système, c'est-à-dire les réseaux de Pétri et dérivés, L.O.T.O.S. décrit un système en définissant les relations temporelles entre les événements possibles du comportement extérieur et observable du système. La structure interne est donc négligée au profit d'un niveau d'abstraction plus adéquat à une telle spécification.

II. Les concepts de base

II.1. Le processus

Le concept de processus correspond à celui de procédure dans les langages de programmation conventionnels. Il nous permet, de cette façon, d'obtenir un principe de structuration puissant et indispensable lors de la spécification de systèmes distribués, un peu comme les réseaux prédicat/transition ou encore, comme le module "corps" dans le langage P.I.P.N.

En L.O.T.O.S, un système est toujours un processus, qui peut lui-même être défini en terme de sous-processus. Sa description se limite à son comportement observable aux points d'interaction.

II.2. Le point d'interaction

Le point d'interaction, comme nous l'avons évoqué dans le chapitre précédent, est généralement connu sous le terme de port de communication. Il s'agit là d'un concept avancé qui remplace les notions traditionnelles d'entrée/sortie et qui permet au processus de communiquer avec son environnement .

II.3. L'ordonnancement temporel

Le principe d'ordonnancement temporel est utilisé pour définir le comportement externe observable d'un processus par la description de l'ordonnancement temporel de ses interactions avec son environnement. Le langage L.O.T.O.S. est , en cela, dérivé de C.C.S. ("Calculus of Communicating Systems") qui fut développé à l'université d'Edimbourg par Milner en 1980.

II.4. Les types abstraits de données

L'utilisation de types abstraits de données renforce l'indépendance requise par rapport à l'implémentation lors de toute spécification et constitue une approche complémentaire non négligeable lors de la spécification de système complexe. En L.O.T.O.S., les structures de données seront décrites dans le langage ACT ONE, développé à l'université de Berlin par Ehrig en 1985.

L.O.T.O.S. repose, donc, sur une approche hybride : processus et type abstrait sur lesquels nous allons nous attarder quelque peu pour bien comprendre la logique sous-jacente à L.O.T.O.S. Ensuite, celui-ci sera abordé en vue d'esquisser une comparaison des différentes techniques de

spécifications évoquées, réseaux de Pétri, P.I.P.N. , L.O.T.O.S.

III. C.C.S. ("Calculus of Communicating Systems")

III.1. Introduction

C.C.S est un autre formalisme permettant de modéliser des systèmes de processus communicants et parallèles. L'une de ses ambitions est de pallier l'absence de constructeur, souvent reprochée aux réseaux de Pétri. Pour ce faire, un système y est représenté comme un ensemble de processus. Un processus se caractérise par :

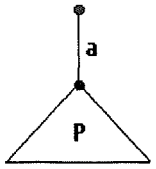
- une liste de points d'interaction ou ports de communication,
- une expression de comportement,
- le domaine des valeurs dans lequel il calcule.

Nous n'envisagerons cependant qu'un sous-ensemble du langage, ce qui nous permettra de mieux en saisir les mécanismes de base. Considérons que la dénomination du port de communication soit également celle de l'observation en ce port. Nous la désignerons sous le vocable d'action atomique observable. Cette simplification peut facilement être levée, comme nous le verrons en L.O.T.O.S, par l'introduction de variables.

Nous donnons ci-dessous une expression de comportement d'un processus C.C.S. :

p = a b nil

où p est un identificateur de comportement et a b nil une expression de comportement. Nil exprime que le processus est inactif tandis que a et b constituent les actions atomiques observables. Soulignons que nil joue un rôle très particulier ; en effet, il est à la fois opérateur et processus. Dès lors, 'b nil ' décrit le processus qui, après réalisation de l'action b, se transforme en un processus inactif (nil). De manière plus générale, l'interprétation de 'a P' peut se faire aisément dans un système de transition :



où P est un système de transition

III.2. Comportements: syntaxe et sémantique

Dans l'expression de comportement ci-dessus, nous avons introduit l'opérateur nil et celui de préfixage dont la syntaxe et la sémantique sont :

- l'inaction : nil

- le préfixage : 'a P '
  où a est une action, P un processus ;
  'a P' signifie qu'après la réalisation de a le processus se transforme en un processus P ; autrement dit :

$$a\ P \xrightarrow{a} P$$

Cependant, il existe plusieurs autres opérations qui peuvent être effectuées entre deux processus :

- le choix : 'P + Q'
  où P et Q sont deux processus
  'P + Q' représente le processus qui se comporte soit comme P soit comme Q

  De sorte, que
  Si P' = a P
  alors 'P+Q' peut effectuer l'action a et se comporter comme P'
  Si Q' = a Q
  alors 'P+Q' peut effectuer l'action a et se comporter comme Q'

  Autrement dit, il y a une perte de choix lors de l'évolution du système ;

- le renommage, qui doit préserver l'observabilité et l'orientation des actions ;

- la restriction 'P \ A'
  où P est un processus
  A un sous-ensemble d'actions atomiques observables
  'P \ A' se comporte comme le processus P pour les actions observables qui n'appartiennent pas à A

- la composition parallèle : 'P|Q'
  où P et Q sont deux processus
  le comportement de 'P|Q' est défini par les règles suivantes :

  * l'entrelacement (ou la non synchronisation)
    Si P = a P_1,
    alors 'P|Q' peut effectuer l'action a, puis se comporter comme $P_1|Q$

    Et de façon symétrique,
    Si Q = b Q_1,
    alors 'P|Q' peut effectuer l'action b, puis se comporter comme $P|Q_1$

  * la synchronisation sur des événements complémentaires (a et $\bar{a}$)

    Si P' = a P et Q' = $\bar{a}$ Q
    alors P|Q peut effectuer une action interne, c'est-à-dire une évolution spontanée, donc non observable et notée ¶, et se comporter comme $P_1|Q_1$.

    Dans ce cas, les deux processus synchronisés sur des actions complémentaires ne peuvent plus communiquer sur l'action de synchronisation.

III.3. Utilité : Axiomatisation de l'équivalence observationnelle

Comme nous avons pu le constater, le langage C.C.S. est très proche des systèmes de transitions. Cependant, il nous permet de vérifier si deux processus sont 'observationnellement' équivalents, c'est-à-dire si aucune observation ne les distingue, et ce, grâce à un ensemble d'axiomes. Citons entre autres, l'idem-potence (x+x = x), la commutativité (x+y = y+x) ou encore le neutre (x+nil = x). Il faut noter que les actions non observables ou internes influencent le comportement observable du système et doivent dès lors être prises en considération. En effet, l'action interne, ¶, est attachée non seulement à la notion de synchronisation, mais également à celle de processus. Tout processus peut évoluer soit de manière interne soit de manière observable tout comme il peut ne pas évoluer. Prenons par exemple deux processus : '¶ b nil + a nil ' et 'a nil + b nil '. Ceux-ci ne sont pas équivalents car il existe une évolution qui amène le premier processus dans un état où seule l'action 'b' est possible. Nous n'aurons donc pas l'axiome ' x = ¶ x ' qui, à première vue, semblait évident. Rappelons que l'ensemble des axiomes doit être prouvé pour démontrer que deux processus sont observationnellement équivalents. Toutefois, si cette démonstration est impossible, cela ne signifie pas qu'ils ne sont pas équivalents.

Cette axiomatisation, liée à la nature du langage, est un atout non négligeable. En effet, si les systèmes de transitions nous permettent eux aussi de vérifier l'équivalence observationnelle, le cheminement suivi, pour ce faire, n'est pas aussi aisé. Il repose sur un algorithme complexe qui se base notamment sur la fermeture du système de transition. Celle-ci permet en effet de représenter les actions atomiques observables pouvant être menées sur le

processus associé. Cette approche, liée aux systèmes de transitions, dépasse cependant le cadre de ce travail. Aussi, renvoyons-nous le lecteur intéressé à [Milner 80].

III.4. Faiblesse : impossibilité de synchronisation multiple

Dans le calcul C.C.S., une synchronisation n'est possible qu'entre deux partenaires car elle donne lieu à une action interne. La synchronisation est ainsi cachée. Cette restriction peut poser, nous semble-t-il, un certain nombre de problèmes particulièrement dans le domaine des protocoles de communication. En effet, il se peut que plusieurs entités doivent, dans l'attente d'une valeur, se synchroniser avec le bus de données.

III.5. Réflexion

Remarquons que si, dans la plupart des cas, il est possible de spécifier le système considéré à l'aide de processus, il n'en reste pas moins vrai qu'une certaine lourdeur commence à apparaître lorsque l'on introduit des structures de données complexes. Cependant, cette approche ne peut être négligée au profit d'une approche par déclaration de types. En effet, dans cette dernière, aucune expression de comportement n'existe à proprement parler comme nous allons le voir dans le langage ACT ONE. Cet aspect statique ne nous permet ni de saisir le comportement attendu du système ni même de le spécifier de manière claire, complète et minimale.

IV. ACT ONE

IV.1. Introduction

ACT ONE est un langage de spécification de types abstraits de données. Ceux-ci se caractérisent par leur absence de référence à l'organisation physique des données, telle qu'on la retrouve dans les langages de programmation conventionnels. Il ne s'agit plus de décrire comment les valeurs sont représentées dans la mémoire ni même de savoir comment certaines procédures vont les manipuler ; mais il faut définir les propriétés et les opérations qui devront être vérifiées pour qu'une implémentation correcte puisse être mise en oeuvre. Un type abstrait de données peut, en conséquence, être considéré comme une spécification formelle d'une classe de types concrets et ce, même s'il nous permet de spécifier bien davantage. Il sera considéré comme un objet mathématique constitué d'ensembles de valeurs, d'opérations et d'axiomes.

IV.2. Syntaxe

Lors de la spécification d'un type abstrait, la première étape consiste à nommer les ensembles de valeurs, désignés sous le vocable sorte, et les opérations (notée : opns) qui leur sont associées. Ainsi, l'ensemble des entiers naturels, "Nat_number", sera défini de la manière suivante :

```
type Nat_number is
 sorts nat
 opns    0    :        -> nat
         succ :  nat -> nat
endtype
```

où 'nat' représente la sorte "nombre naturel" et '0' et 'succ' les opérations qui lui sont associées. L'opération 'succ' permet, à partir d'un élément de la sorte 'nat', d'obtenir son successeur. Celui-ci sera de même sorte que l'argument de l'opération soit un nombre naturel. L'opération '0', quant à elle, n'a pas d'argument et est donc une constante. Elle signifie que zéro est un naturel.

La définition des sortes et des opérations est appelée la signature du type de données. Elle nous permet non seulement d'en définir la syntaxe mais également d'en représenter tous les éléments. Cependant, il est parfois souhaitable d'avoir à notre disposition des opérations supplémentaires dont l'interprétation, pour être correcte, ne peut se faire que dans un contexte bien précis. Ce dernier englobe les propriétés des opérations et est défini par un ensemble d'axiomes. Pensons à l'opération d'addition entre deux nombres naturels dans laquelle, par exemple, l'addition d'un nombre quelconque 'x' avec l'élément nul nous redonne ce même nombre 'x' et ce pour tout 'x'. La définition correcte de l'opération d'addition se fera dès lors de la manière suivante :

```
eqns
  forall   x,y : nat
  ofsort   nat
    x + 0          =     x ;
     x + succ(y)   =      succ(x+y);
```

où 'ofsort' indique la sorte des expressions intervenant dans l'équation. La définition complète de l'ensemble des entiers naturels devient :

```
type Nat_number  is
     sorts   nat
     opns    0   :                -> nat
             succ : nat -> nat
             _+_  : nat,nat       -> nat;
 eqns   forall    x,y : nat
```

```
    ofsort    nat
      x+0             = x ;
      x+succ(y)       = succ(x + y);
endtype
```

où '_' indique la place des arguments. Ce type est défini comme tel dans une librairie prédéfinie. D'autres types de base, tels que le type booléen, bit ou octet, peuvent y être retrouvés. Notons qu'ils peuvent être importés dans la spécification par une simple requête.

IV.3. Construction hiérarchique des types de données

ACT ONE permet de compléter des types de données existant tant par l'ajout de sortes que celui d'opérations et d'équations. Ainsi, la notion d'identificateur utilisée lors de la spécification de la couche liaison de données de Fip a-t-elle été construite sur le type 'Boolean', de la manière suivante :

```
type  Identificateur  is  Boolean
 sorts Ident
 opns    prod     :-> Ident
         cons     :  -> Ident
eqns forall     x : Ident
       ofsort     Bool
         x  eq  x      = true  ;
         prod eq cons = false ;
         cons eq prod = false ;
endtype
```

Ce type nous permettra entre autre de vérifier si l'identificateur produit est celui qui est ou non consommé par l'entité concernée. En général, nous pouvons combiner plusieurs définitions de types simultanément. Citons pour mémoire l'opération de création d'un buffer associé à un identificateur et qui aura pour arguments une donnée, le nom d'un buffer et celui d'un identificateur.

ACT ONE nous permet également de spécifier des types paramétrés. Citons par exemple une 'queue' actualisée à une queue d'entiers ou de caractères. Ceci nous permet d'éviter une duplication notamment au niveau des équations. En effet, que la queue soit faite d'entiers ou de caractères, l'obtention de son premier élément ('premier') est définie de manière unique :

premier : queue -> nat

<---> premier : queue -> élément

premier : queue -> char

Pour ce faire, la sorte formelle 'élément' a été introduite, ce qui revient à spécifier une queue d'un élément générique.

```
type Queue is
 formalsorts  élément
 formalopns   e0:          -> élément
 sorts        queue
 opns         créer :                          -> queue
              ajouter : élément,queue          -> queue
              premier : queue                  -> élément
 eqns
  forall   x,y : élément,   z : queue
  ofsort   élément
  premier(créer) = e0;
  premier(ajouter(x ,créer)) = x ;
  premier(ajouter(x,ajouter(y,z))) = premier(ajouter(y,z));
endtype
```

Celui-ci pourra être actualisé, par exemple à une queue d'entiers, de la manière suivante :

```
type Nat_number_queue is  Queue
      actualizedby  Nat_number  using
      sortnames nat   for   élément
      opnnames   0          for   e0
endtype
```

Remarquons au passage que "la sémantique associée au type de donnée 'queue' est la même que la sémantique qui serait associée à celui de connexion" [ISO 89]. En effet, l'opération d'obtention du premier élément de la liste peut être associée à la réception d'un message, tout comme l'opération d'addition peut l'être à l'envoi du message. Il suffit dès lors de procéder au renomage des sortes et des opérations du type de données considéré.

```
type connexion is Queue renamedby
 sortnames canal          for    queue
                objet      for    élément
 opnnames recevoir  for           ajouter
             envoyer      for    premier
endtype
```

IV.4. Réflexion

Le langage ACT ONE ne nous permet de décrire que le niveau fonctionnel d'un système et ce à l'aide des domaines de valeurs et des fonctions sur ces derniers. En ce sens, sa notion de type abstrait diffère de celle de classe présente dans les langages orientés objets. Un type abstrait, décrit dans le langage ACT ONE, n'existe pas par lui-même et ses opérations ne revêtent en aucun cas un caractère dynamique. Ainsi l'approche hybride adoptée par L.O.T.O.S. est justifiée et lui permet d'acquérir des qualités supplémentaires tant au niveau de la maintenabilité que de le réemploi.

Notons, au passage, que la description des données aurait tout aussi bien pu se faire dans un autre langage de spécification de types abstraits.

V. L.O.T.O.S. "Language of Temporal Ordering Specification"

V.1. De C.C.S. à L.O.T.O.S.

L.O.T.O.S. peut être considéré comme l'une des variantes du langage C.C.S. dans laquelle l'opérateur de composition parallèle s'est quelque peu modifié. La synchronisation n'est plus cachée ; tout comme la composition, cette intention constitue désormais un opérateur en soi (hiding). Cette distinction permet à plusieurs processus de se synchroniser sur un même événement puisque les actions de synchronisation restent offertes à l'environnement.

Dans la syntaxe des expressions de comportement, cette modification n'est cependant pas unique comme nous l'indique le tableau ci-dessous.

	C.C.S.	L.O.T.O.S.
- Inaction	'nil'	'stop'
- Préfixage	a P	a ; P
- Choix	P + Q	P [] Q
- Composition parallèle	P | Q	P |[a_1 .. a_n]| Q
- 'Hiding'		'Hide' a 'in'
- Restriction	\ {a}	

En outre, elle a entraîné l'apparition de nouveaux opérateurs tels que la synchronisation séquentielle ou encore le parallélisme avec ou sans interaction.

Avant d'aller plus loin, levons l'hypothèse émise lors de l'introduction à C.C.S. : la dénomination du port de communication n'est plus nécessairement l'observation en ce port. Ainsi, lorsque le processus 'arbitre de bus' envoie un identificateur par le point d'interaction g, il s'attend à en recevoir la valeur sur le même port. C'est notamment à ce niveau qu'intervient le langage ACT ONE. En effet, il ne connaît pas la valeur liée à l'identificateur qu'il a émis sur le bus mais sait uniquement qu'il doit recevoir un élément d'un certain type (soit Data) et qu'il devra l'interpréter comme cette valeur. Ces interactions, émission et réception, seront modélisées de la façon suivante :

- émission :

  g !Identificateur — où Identificateur désigne l'identificateur offert

- réception

  g ?x : Data — où x représente la variable de type 'Data' dont la valeur ne sera fixée qu'après l'événement.

De la même façon, lorsque l'arbitre de bus a diffusé l'identificateur sur le bus, celui-ci est reçu par les différentes entités qui sont connectées à ce bus, et ce après synchronisation.

V.2. La Synchronisation entre processus

Un processus se synchronise avec son environnement par une offre d'événement (ou d'interaction). Cette offre, par un point d'interaction g, peut comporter plusieurs attributs. Ceux-ci, répartis en deux types, peuvent correspondre soit à une déclaration de valeur (g !Identificateur) soit à une déclaration de variable (g ?x : Data). Notons toutefois, l'impossibilité en L.O.T.O.S. d'exprimer des structures de variables telles que le 'record' en Pascal. Cette restriction nous oblige à définir une requête, par exemple 'Rpdat(val)', à l'aide de constantes pour désigner les primitives (Rpdat, ...) et les variables d'un certain type (val de type 'Data').

Le processus 'Arbitre de bus' peut de cette manière offrir l'interaction suivante :

g ! Rpdat ? val : Data

où ! Rpdat revêt un caractère particulier. Selon notre convention, cela revient à dire que ce processus enverrait une requête 'Rpdat'. Il n'en est rien ; l'arbitre sait qu'après avoir envoyé une requête 'Iddat(identificateur)',il doit recevoir la valeur liée à cet identificateur par l'entité productrice de celui-ci et ce,via la requête 'Rpdat(valeur)' ; par exemple :

g ! Rpdat ! dat_c où 'dat_c'

représente la valeur demandée. Dans ce cas, les deux processus offrent la même valeur de primitive au même point d'interaction, ce qui leur permettra, si 'dat_c' est de type 'Data', de se synchroniser. Nous venons d'évoquer deux types

de synchronisation : la première, dite dure ou de type rendez-vous où le processus et l'environnement offrent la même valeur au même point d'interaction (g ! Rpdat) ; la seconde qui peut être interprétée comme un passage de valeur. En effet, si le processus offre g ?x : Data et l'environnement g !dat_c, ou l'inverse, pour autant que dat_c soit de type 'Data', la valeur de x après l'événement sera la seule valeur qui ait permis cet événement.

Enfin, si le processus offre g ?x : Data et l'environnement g ?y : Data, ils pourront également se synchroniser. Dans ce cas, les valeurs de x et y seront identiques mais quelconques puisque toute valeur de type 'Data' permet l'événement. Nous parlerons plus volontiers de génération de valeurs.

Synthétisons ces différentes notions par le tableau suivant:

Processus A	Processus B	Condition de synchronisation	Type d'interaction	effet
g !E_1	g !E_2	valeur de E_1 = valeur de E_2	Egalité symbole par symbole ('matching')	synchronisation
g !E	g ?x : t	valeur de E est de sorte t	passage de valeur	après synchronisation : x = valeur de E
g ?x : t	g ?y : u	t = u	génération de valeur	après synchronisation : x = y = v où v est une valeur de sorte t

Rappelons, cependant, que les offres ne seront compatibles que si elles référencent le même point d'interaction, présentent le même nombre d'attributs et permetent l'une des trois synchronisations précitées.

Toutefois, certaines offres ne doivent pas être prises en compte par certains processus. Ainsi, lorsque le processus 'Arbitre de bus' envoie un identificateur à scruter, seul le producteur de cet identificateur devra tenir compte de la requête puisque lui seul est habilité à en renvoyer la valeur sur le bus. Pour ce faire, un prédicat sera ajouté à l'offre d'événement. S'il est vérifié, l'événement sera autorisé ; s'il ne l'est, il ne sera pas pris en considération. Dans le cas précédent, l'offre d'interaction se traduira de la manière suivante :

g ! Iddat ? id : Ident [producteur(nom, id)]

où 'nom' désigne l'entité concernée à savoir, soit la station soit le répondeur,
'id' est l'identificateur sur lequel est véhiculé la demande,
[producteur(nom,id)] permet de vérifier que l'entité concernée est ou non celle qui produit 'id'

En L.O.T.OS., les expressions de comportements sont construites à partir de ces offres d'événement et d'autres expressions de comportement.

V.3. Les expressions de comportement

Comme nous l'avons souligné lors de l'introduction à C.C.S., ces expressions vont permettre de définir l'ordre dans lequel le processus pourra participer aux événements offerts

par son environnement aux différents points d'interaction. Remarquons que L.O.T.O.S. nous permet de paramétrer la définition d'un processus, non seulement en terme de ports formels de communication, mais également en terme de liste paramétrée. Par exemple, le processus 'Arbitre de bus' sera défini de la manière suivante :

```
process  Arbitre [g] (l : liste, s : état) : noexit :=  ...
                                :
                            endprocess
```

où g est le port de communication,
l, s : deux paramètres (qui font intervenir les types abstraits 'liste' et 'état'),
'noexit' est la fonctionnalité du processus et se réfère à sa terminaison, comme nous le verrons dans la sous-section V.4.1. consacrée à la composition séquentielle.

Cette définition formelle nous permet notamment, à partir de l'opérateur de préfixage, d'obtenir l'expression de comportement suivante :

```
g ! Rpdat  ? val : Data  ;  Arbitre [g]  (l, Next_per)
```

où Arbitre [g] (l,Next_per) représente une instanciation du processus 'Arbitre de bus' et où g ! Rpdat ?val : Data n'est rien d'autre que l'action ou offre d'événement. L'expression, prise dans son ensemble, signifie que dès que l'arbitre de bus aura reçu la valeur de l'identificateur via la requête 'Rpdat(valeur)', il passera à l'identificateur périodique suivant ('Next_per') dans la liste à scruter ('l'). Cependant, cette expression n'est pas la seule envisageable

puisque l'arbitre de bus doit, au préalable, avoir envoyé un identificateur sur le bus. Il doit également, lorsque la liste des identificateurs est vide, la réinitialiser. Pour ce faire, il est nécessaire d'introduire le concept d'événement interne i et celui de garde. L'événement interne n'est pas observable tel quel. Il permet au processus, à l'insu de son environnement, de réinitialiser la liste à scruter pour sastisfaire à la contrainte de périodicité. Cet élément interne a alors des conséquences (observables) sur le comportement futur du processus l'ayant effectué. Ce sont là les raisons sous-jacentes à sa prise en considération et qui ont amené le concepteur à ne définir qu'un seul élément interne, noté i . Rappelons, en effet, que selon le principe d'observabilité, il est impossible de différencier des événements internes. Le concept de garde, quant à lui, rejoint celui de prédicat ajouté à une offre d'événement. Cependant, dans ce cas, le prédicat ne porte plus sur les éléments reçus lors de l'interaction mais bien sur ceux connus à ce moment là. Ces éléments sont décisifs dans la réalisation de l'interaction. Par exemple, dans le cas précédent, le fait que la liste soit vide ou non est indépendant de l'offre d'événement 'Iddat(identificateur)' mais elle en interdit la réalisation s'il n'y a plus d'identificateur à scruter.

Le processus 'Arbitre de bus' se comportera comme l'un de ces trois sous-processus : envoi de Iddat, réception de Rpdat et réinitialisation de la liste périodique à scruter. Cependant, si la logique sous-jacente est ici le choix, il n'en va pas toujours de même. Deux sous-processus peuvent s'effectuer séquentiellement, voire même en parallèle.

V.4. La description en terme de sous-processus

V.4.1. La composition séquentielle

Supposons que nous ayons deux processus : P et Q ; il n'est pas rare que, pour se réaliser, le processus Q doive attendre que P se soit terminé correctement. Autrement dit, nous dirons que le processus Q est activé par le processus P, ce qui se notera : 'P >> Q'. Aussi, pour désigner l'endroit où le processus P se termine avec succès, nous utiliserons le processus exit. Il s'agit là d'une alternative au processus stop qui permet de transférer le contrôle au second processus. Cependant, il se peut que pour s'exécuter le processus Q ait besoin d'informations issues du processus P. Dans ce cas, le processus exit sera caractérisé par la liste de ces informations ou valeurs. Le processus Q, pour les utiliser, devra définir des variables qui pourront les contenir et ce via la construction 'accept x1 : t1, x2 : t2 ... xn : tn in' , où les xi sont les variables en question, et les ti, leur type. Nous aurons dès lors ,

```
P >> accept  x1 : t1, x2 : t2 ... xn : tn in  Q
```

Soulignons que la liste des types des valeurs que transmet P à Q s'appelle la fonctionalité de P. Aussi, les types intervenant dans la construction accept doivent ils correspondre à cette fonctionnalité pour que le relais entre P et Q puisse avoir lieu. La fonctionalité doit être déclarée lors de la définition du processus. Par définition, celle du processus exit vaut 'exit', et celle du processus stop vaut 'noexit'.

Si un processus peut, comme nous venons de le voir, être activé par un autre processus, il se peut aussi qu'il soit désactivé : Q [> P. Cette notation signifie que tant que le processus P ne démarre pas, 'Q [> P' se comporte comme le processus Q. Toutefois, dès que P s'exécute, le processus Q s'arrête. Il se peut donc qu'il ne soit jamais totalement exécuté. Inversement, si le processus Q se termine avec succès avant le démarrage de P, ce dernier ne sera plus possible.

V.4.2. Le parallélisme

Deux cas de parallélisme sont à distinguer : celui sans interaction et celui avec interaction. Pour les expliciter, considérons notre processus Fip. Celui-ci se compose de quatre sous-processus : l'arbitre de bus, le répondeur, la station et le médium (également appelé 'bus'). Les trois premiers sont des processus indépendants. Ils évoluent librement et leurs échanges d'information ne s'effectuent que par leur synchronisation avec le médium.

La première forme de parallélisme, soit sans interaction, est sous-jacente aux comportements de l'arbitre de bus, du répondeur et de la station et se définit par l'opérateur ' |||'. Le processus de comportement :

arbitre de bus ||| répondeur ||| station

offre à l'environnement les événements de chacun de ces sous-processus. Ces derniers ne peuvent en aucun cas se synchroniser entre eux sur base de ces événements. Supposons qu'une offre d'événement du processus arbitre et une du processus station soient satisfaites. L'un ou l'autre des

deux processus peut évoluer avec son environnement et ce, en vertu du principe d'atomicité des actions : "deux événements ne peuvent avoir lieu en même temps, même s'ils sont indépendants un seul évoluera à la fois" [Lotos 89]. L'offre d'interaction du second processus subsiste et peut ensuite se concrétiser. Nous sommes alors confrontés à l'entrelacement des comportements de l'arbitre de bus et de la station et ce, en respect des relations d'ordre pour chacun d'eux (comparer à 'arbitre[] station').

La deuxième forme de parallélisme, soit avec interaction, est celle qui régit notamment le comportement de la station et du médium. Ces deux dernières entitiés doivent, pour pouvoir s'exécuter, interagir sur les ports de communication 'fromeds' et 'tomeds'. Ces ports sont également appelés 'port de synchronisation'. En effet, pour évoluer, les deux processus doivent offrir le même événement sur le même port de communication : soit 'fromeds' soit 'tomeds'. Cet opérateur de parallélisme se note :

station |[fromeds, tomeds]| medium

Supposons que la station offre un événement par un point d'interaction différent que ces deux ports de synchronisation comme par exemple une confirmation de mise à jour de valeur (par la requête l_put_conf (id)) sur son point d'interaction vers la couche application ('applic_s') ; dans ce cas, le processus de comportement 'station |[fromeds, tomeds]| medium' offre également cet événement . Tout événement se produisant en dehors de 'fromeds' et 'tomeds' ne fait évoluer qu'un seul sous-processus (ici, la station). Cette possibilité n'est cependant pas offerte lorsqu'on envisage le parallélisme avec interaction sur tous les ports de communication. Dans ce cas, toute action de la station doit être synchronisée avec une action du médium, et vice versa. Cette synchronisation est

très sévère et n'est pas envisageable dans le cadre de notre application. Sa forme syntaxique est la suivante : 'station || médium'.

La synchronisation des différentes entités ('station', 'répondeur' et 'arbitre de bus') avec le 'medium' dépend dès lors de l'environnement. Pourtant, il semble assez logique, dans le cas qui nous occupe, de nous situer à un niveau d'abstraction supérieur, c'est à dire de ne percevoir du système que son comportement en tant que le reflet de la spécification en langage naturel, autrement dit, le séquencement des requêtes (Iddat, Rpdat, ..) et le service rendu à la couche application (l_put_conf,...). Il est, en effet, inutile de percevoir les synchronisations sur le 'medium'. Pour ce faire, nous utilisons l'opérateur de restriction hiding sur les ports qui permettent aux entités 'arbitre de bus', 'répondeur' et 'station' de se synchroniser avec le médium. Cet opérateur transforme ces ports de communication en ports internes. En conséquence, toutes les offres d'événements faites sur ces ports deviennent des actions non observables ou internes. Rappelons, que ce dernier concept est sous-jacent à celui d'équivalence observationnelle que nous avons évoqué lors de l'introduction à C.C.S. Si celle-ci peut être appliquée, il existe également en L.O.T.O.S. un certain nombre de propriétés, ou règle de réécriture, qui permettent de remplacer certaines sous-expressions par des expressions équivalentes. Leurs définitions formelles est donnée dans [LOTOS 89]. Notons toutefois, que la notion d'équivalence faisant intervenir à la fois les types de données et les processus n'est pas formalisée, bien qu'elle soit fondamentale.

Sur base de ces différents concepts, voyons comment nous avons procédé à l'élaboration de la spécification du service périodique de Fip en L.O.T.O.S.

V.5. Procédé d'élaboration de la spécification LOTOS

L'objectif de notre travail était, faut-il le rappeler, de spécifier la couche liaison de données de Fip en L.O.T.O.S. en nous basant sur le modèle des réseaux de Pétri élaboré par Nordgard et du modèle décrit dans le langage P.I.P.N. Aussi, notre première idée a-t-elle consisté à produire une spécification structurée, basée sur la notion d'états et de transitions.

La décomposition du processus Fip s'est calquée sur celle du modèle P.I.P.N. et nous y trouvons les processus 'arbitre du bus', 'répondeur', 'station' et 'médium'. De plus, les opérateurs L.O.T.O.S. de synchronisation de deux processus traduisent la sémantique de composition de modules P.I.P.N. , qui, rappelons-le, est-elle même inspirée de C.C.S. (tout comme L.O.T.O.S.). Aussi, la traduction en était-elle assez triviale.

Au sein d'un processus, les opérateurs traduisent, par contre, la sémantique tant des réseaux de Pétri que de ceux prédicats/transitions. Citons l'opérateur de choix qui correspond à une place partagée entre deux transitions en conflit ou encore l'instanciation d'un processus qui décrit le marquage initial. Dès lors, nous avons simulé la notion d'état à l'aide d'une variable qui est en fait un paramètre formel 's' de type 'Etat'. De cette manière, nous avons pour le processus 'station' :

```
process Station [fromed, tomed, applic] (s : Etat, ...) :
noexit :=
                                    :
                                endprocess
```

Après son instanciation, nous vérifions, à l'aide d'une garde, que nous sommes bien dans l'état à partir duquel nous pouvons tirer la transition qui y est associée. Lorsqu'elle est à l'état repos ('idle'), la station par exemple peut aussi bien recevoir une demande de mise à jour ou de transfert de fichier qu'une demande d'identification de variable. La réalisation de chacun de ces événements amène la station dans un autre état avec un appel de processus effectué avec la valeur instanciée en paramètre. Nous trouvons, ci-dessous, une implémentation de ce procédé.

```
[s eq Idle]
   ->(
        (applic  !L_put_dem  ?id  :  Ident  ?val1  :  Data
        [producteur(nom,id)] ;
               station [fromed,tomed,applic] (Put,...)
        )
     []
        (applic !L_get_dem ?id:Ident [consommateur(nom,id)] ;
                station [fromed,tomed,applic] (Get, ...)
        )
     []
        (fromed !Iddat ?id : Ident  [producteur(nom,id)];
                station [fromed,tomed,applic] (Send, ...)
         []
         fromed !Iddat ?id : Ident [consommateur(nom,id)];
                station [fromed,tomed,applic] (Read, ...)
         )
    )
```

Une telle description est assez lourde. Non seulement sa lecture n'est pas aisée mais on constate dans sa partie de description de données, la sémantique fort contraignante du type de donnée 'Etat'. Il est nécessaire de définir les propriétés de l'opération de comparaison 'eq' qui nous permet de vérifier l'état dans lequel nous sommes. Cette opération nécessite une énumération complète des états distincts deux par deux mais elle n'est envisageable que si leur nombre n'est pas trop élevé. Ces inconvénients ont mis en relief l'inadéquation d'une telle approche dans le langage L.O.T.O.S.

Aussi, dans le cadre d'une seconde approche, nous sommes nous davantage basés sur les caractéristiques propres à ce langage ; à savoir l'ordonnancement temporel. Nous nous sommes inspirés de la séquentialité des transitions du modèle P.I.P.N. , tout en faisant abstraction de l'état interne du système. A une séquence de transitions correspond désormais une seule expression de comportement. Cette expression résulte de l'entrelacement des diverses transitions appartenant à la séquence envisagée. Toutefois, notons que les états intermédiaires peuvent être facilement déduits de la spécification ainsi obtenue. En effet, supposons que l'arbitre de bus ait envoyé la requête d'identification sur le premier élément de la liste périodique à scruter (med !Iddat !first(l)) ; il passe implicitement dans un état d'attente puisqu'il se synchronise avec l'événement ' med !Rpdat ?val:Data '. Lorsque celui-ci se produira, l'arbitre de bus passera à l'identification de l'élément suivant. Nous décrivons ce processus de la manière suivante :

```
med !Iddat  !first(l) ;
(med !Rpdat ?val : Data ; Arbitre [med] (tail(l,first(l))))
```

Cette approche, fort proche de la spécification en langage naturel, reflète mieux les services attendus par la couche liaison de données. Seuls ses notations et mécanismes de synchronisation nous semblent assez rébarbatifs pour un lecteur non averti. Comme nous le verrons au chapitre suivant, l'attitude du lecteur sera fonction de multiples facteurs comme par exemple les outils et les exemples mis à sa disposition. Dans le cadre de L.O.T.O.S., ces divers aspects ont été pris en compte dans le cadre du projet 'Software Environment for the Design of Open Distributed Systems' (S.E.D.O.S.)

VI. L.O.T.O.S et le projet S.E.DO.S

L.O.T.O.S., rappelons-le, est une technique de description formelle qui a été développée pour la spécification de systèmes distribués et qui est devenue, depuis peu, une norme standard, IS 8807. Cette norme a obligé ses concepteurs à démontrer son adéquation aux services et aux protocoles du modèle OSI, à promouvoir sa définition et à développer des outils propres à aider le concepteur dès la spécification du système jusqu'à son implémentation et sa vérification. C'est à cette fin qu'a été mis sur pied en 1984 le projet S.E.D.O.S. Ce projet était, en effet, consacré au développement et au soutien de deux techniques de description formelles Estelle et L.O.T.O.S. et a impliqué plus de dix organistions au travers six pays jusqu'en 1988 [SEDOS 88].

La partie du projet relative à L.O.T.O.S. comprenait trois tâches distinctes : "langage et spécification", "vérification" et "outils". Nous allons en voir successivement les diverses implications [LOTOS 89].

VI.1. "Langage et spécification"

Cette tâche était responsable des propositions d'améliorations du langage et de l'application de L.O.T.O.S. en tant que technique de description formelle. L'une des contributions les plus importantes à l'amélioration du langage est la possibilité offerte à plus de deux processus de se synchroniser sur un même événement. De telles contributions se sont accompagnées de spécifications "d'essai" de protocoles et de services du modèle OSI [SEDOS 88]. Ces spécifications remplacent une évaluation motivée du pouvoir d'expression de L.O.T.O.S.

VI.2. "Vérification" ou Développement de spécifications "d'essai"

Le développement de ces spécifications avait pour objectifs d'une part de montrer quels étaient les effets qui pouvaient être attendus de l'application d'une technique de description formelle et d'autre part de prouver que le pouvoir d'expression et le degré d'abstraction d'une telle technique ne pouvaient être adéquatement évalués que sur base d'expériences issues du monde réel. Dans cette perspective, L.O.T.O.S. a été appliqué au protocole de transport, à HDLC et à bien d'autres exemples issus principalement du modèle OSI [LOTOS 89]. Ces expériences ont montré combien il était difficile de décrire complètement un standard. Cette difficulté réside non seulement dans la complexité du système envisagé mais également dans le souci de préserver la clareté et la structure naturelle de la description.

VI.3. "Outils"

Le développement des outils qui gravitent autour de L.O.T.O.S. n'a pas été facile. En effet, le langage et les outils, actuellement disponibles, ont dû être développés en parallèle afin non seulement de disposer d'outils lors de l'adoption de L.O.T.O.S. en tant que standard mais également en vue d'obtenir un langage qui soit approprié à la production d'outils. Citons MELO ("Mentor LOTOS Editor") et BELASI ("Behavioural Language Simulator") qui furent très rapidement disponibles [SEDOS 88]. Quant à nous, nous avons utilisé le simulateur HIPPO qui contient, entre autre, un vérificateur syntaxique (SCLOTOS) et un outil qui réalise, notamment, l'insertion des types abstraits de la librairie (LISA). Cet outil nous a permis de simuler notre description et ainsi de

vérifier si elle correspondait bien au résultat fourni lors du prototypage rapide du modèle P.I.P.N.

Notons cependant que le développement des outils étant fort récent, de nombreux efforts sont faits ici et là en vue d'obtenir, par exemple, des vérificateurs qui ne soient plus limités à l'équivalence observationnelle et qui permettraient d'effectuer des tests sur la spécification obtenue (conformité, robustesse..). Des travaux sont effectués en ce sens depuis le début de cette année au L.A.A.S. D'autres travaux tentent de rendre les outils existants plus performants ou de traduire la description obtenue en un système de transitions labellés.

VII. Conclusion

L.O.T.O.S. est un langage normalisé qui permet de spécifier des systèmes distribués complexes et sur lequel se greffe déjà un ensemble d'outils de support (éditeurs, simulateurs,..). Le manque de temps et l'absence de vérification autre qu'observationnelle sont cependant des lacunes qui font aujourd'hui l'objet d'une étude plus approfondie et qui sont liées au caractère récent de ce langage. Son accession rapide au statut de standard et les résultats déjà atteints nous permettent d'entrevoir en lui un langage clé pour la spécification de systèmes distribués dans l'avenir. Notons toutefois, que son pouvoir d'expression nous a permis de constater son adéquation à une spécification claire, relativement naturelle et concise du protocole Fip, ce qui n'était pas le cas pour les spécifications sur lesquelles nous nous sommes basée.

Chapitre 4 : Les réseaux de Pétri, P.I.P.N., L.O.T.O.S. - ou trois approches distinctes

I. Introduction

Tout au long de ce travail, nous avons présenté les trois formalismes, que sont les réseaux de Pétri, les langages P.I.P.N. et L.O.T.O.S., en tant que support pour la spécification de protocoles. Cependant, nous nous devons ici de nuancer notre approche. En effet, si la spécification définit quelles sont les propriétés attendues du système à concevoir, elle peut être décomposée en une hiérarchie plus fine de spécifications lors de la conception du protocole. En conséquence, le formalisme utilisé se doit d'être adapté au niveau d'abstraction requis. Le choix du formalisme est cependant loin d'être aisé, car il dépendra également des connaissances ou de l'objectif poursuivi par le concepteur (pédagogique, expérimentatif ..), ou encore du temps imparti pour la réalisation du projet en cours.

Au cours de ce chapitre, nous allons mettre en évidence les différences fondamentales qui existent entre les trois formalismes sur base desquels s'est réalisé notre travail, à savoir les réseaux de Pétri, P.I.P.N. et L.O.T.O.S. Nous donnerons ensuite quelques indications quant à l'utilisation qui peut en être faite. Soulignons cependant, que cette comparaison est le fruit d'une réflexion basée sur l'application de ces différents formalismes au service périodique de la couche liaison de données de Fip et qu'elle mériterait une analyse non seulement plus large mais également plus approfondie.

II. Formalismes de spécification et protocoles

II.1. Spécification dans le domaine des protocoles de communication

Dans un premier temps, il est nécessaire de clarifier la signification de la spécification dans le domaine des protocoles de communication. Comme nous l'avons déjà évoqué au chapitre 1, l'architecture de communication d'un système distribué, tel que Fip, est structurée comme une hiérarchie de couches. Chaque couche offre un ensemble de services à ses utilisateurs. Ceux-ci ne sont concernés que par la nature du service qui leur est offert. Ce comportement, en entrée et en sortie de la couche de protocole, en définit la spécification de service. Cette spécification est généralement basée sur un ensemble de primitives de service qui doivent être éxécutées de manière non arbitraire. Pour mémoire, la couche liaison de données de Fip offre notamment les primitives 'L_put_conf', l_received_ind' (...) à la couche application. Cependant, si cette spécification est très utile, elle reste néanmoins insuffisante pour décrire le protocole dans son ensemble et doit être raffinée. La couche de protocole doit être décomposée en entités qui communiqueront entre elles par les services de la couche inférieure. La description de l'ensemble des opérations propres à chaque entité correspond à la spécification du protocole.

Cette distinction est fondamentale et se reflète non seulement, au niveau du cycle de conception d'un protocole mais également dans le choix du formalisme de spécification. L.O.T.O.S. , par exemple, semble particulièrement bien adapté à la spécification de services. Ceux-ci correspondent effectivement au comportement observable d'un empilement de couches de protocole tel qu'il est vu par ses utilisateurs.

II.2. Cycle de conception d'un protocole et formalismes de spécification

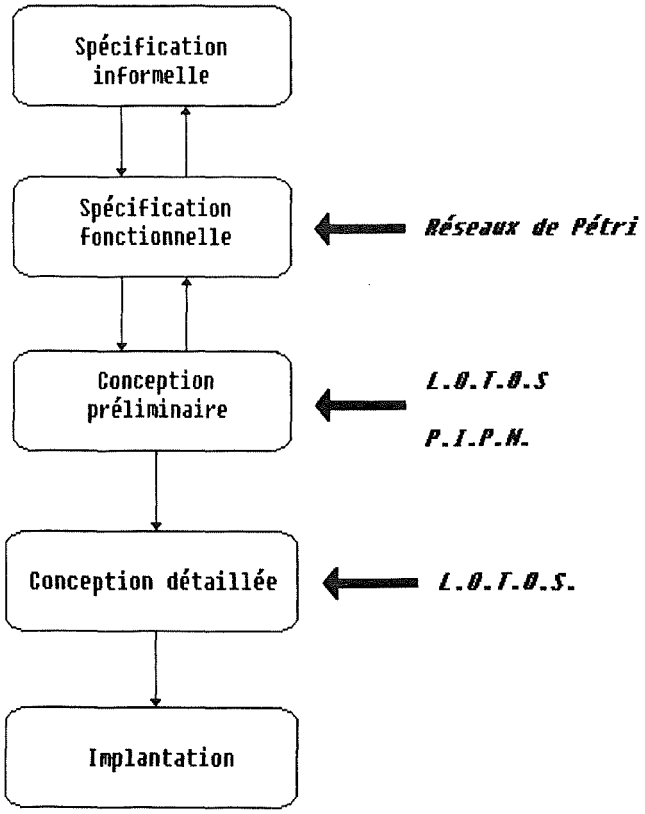
La conception d'un protocole est similaire à celle d'un logiciel et se décompose en cinq étapes distinctes : la spécification informelle, la spécification fonctionnelle, la conception préliminaire, la conception détaillée et l'implantation.

La spécification informelle est généralement rédigée en langage naturel et s'attache à la description complète du protocole. C'est celle que l'on retrouve, notamment, dans les publications de l'ISO. La spécification fonctionnelle, quant à elle, décrit les activités observables que doit réaliser le système. Elle définit dès lors, pour tout protocole, les services en terme de primitives ; celles-ci peuvent être décrites en terme de système de transitions, ou de réseaux de Pétri tels ceux élaborés par Nordgard (Annexe A). De manière plus générale, cette spécification de services sert de référence formelle pour la suite des travaux.

La conception préliminaire permet de structurer le système en mettant l'accent soit sur les données, soit sur les processus. Cette dernière technique de décomposition est adoptée aussi bien par le langage P.I.P.N. que par le langage L.O.T.O.S. ; elle est synonyme de décomposition modulaire sur base de laquelle peut être obtenue une description procédurale qui constitue l'étape de conception détaillée. Cette dernière étape peut être obtenue systématiquement lorsqu'il existe des outils de génération automatique de code comme c'est le cas pour L.O.T.O.S..

Enfin, la dernière étape est celle de l'implantation du système qui souligne l'intérêt d'un langage normalisé tel que L.O.T.O.S..

L'ensemble de ces étapes constitue le cycle de conception d'un protocole et peut être représenté de la manière suivante :



Ce schéma nous montre aisément que les réseaux de Pétri ne se situent pas au même niveau que L.O.T.O.S. ou P.I.P.N. Ceci tient également au fait que les réseaux de Pétri servent notamment de référence sémantique tant au langage P.I.P.N. que L.O.T.O.S. Cependant, étant donné la pauvreté d'expression des réseaux de Pétri, il ne nous semble pas correct de comparer ces trois techniques. Nous envisagerons les réseaux prédicats/transitions qui sont plus proches des deux langages envisagés même s'ils se situent à un niveau identique à celui des réseaux de Pétri.

III . Caractéristiques générales des réseaux prédicats/transitions, P.I.P.N. et L.O.T.O.S

Afin de mettre en valeur les différences principales qui existent entre les trois formalismes de spécification sur lesquels repose notre travail, nous avons dressé un tableau comparatif sur base d'un certain nombre de propriétés associées au système envisagé. Celles-ci sont, selon nous, essentielles pour une bonne perception d'ensemble et reposent principalement sur l'aspect théorique de ces formalismes. Nous en avons retenu huit :

- l'approche adoptée : structurée, non structurée
- l'axe de la description : transitions, types abstraits,..
- la sémantique du langage
- les bases théoriques sous-jacentes à ce formalisme : C.C.S,...
- les mécanismes de synchronisation rencontrés
- les techniques d'analyse envisageables
- les types d'application possibles : modélisation, vérification,..
- les limites

Ces diverses caractéristiques ne représentent qu'un noyau de l'ensemble des propriétés envisageables. Nous aurions pu notamment prendre en considération les aspects de parallélisme et de concurrence. Ceux-ci recouvrant partiellement les mécanismes de synchronisation, nous n'évoquerons ces deux aspects que lors des commentaires associés au tableau et ce, au niveau des mécanismes de synchronisation. De manière similaire, le degré d'abstraction permis par le langage recoupe non seulement le type d'approche autorisé mais également l'une des bases théoriques sur laquelle repose la

description du système ; il sera donc évoqué à ces différents niveaux du commentaire.

Présentons le tableau suivi des différents commentaires qu'il suscite.

	Réseaux prédicats/transitions	P.I.P.N.	L.O.T.O.S.
Approche	non structurée	structurée (modules)	structurée (processus et types abstraits)
Description axée sur	transitions et états	transitions et unification	transitions et types abstraits
Sémantique	réseaux de Pétri et transition indépendante	réseaux prédicats/transitions avec rendez-vous	opérationnelle basée sur les systèmes de transitions entrelacées
Base théorique	logique du premier ordre	logique du premier ordre et C.C.S.	C.C.S. et ACT ONE
Mécanismes de synchronisation	/	rendez-vous et F.I.F.O	rendez-vous
Techniques d'analyse	par invariant et par énumération	par invariant et par énumération	algébrique (équivalence de comportement)
Application	modélisation et vérification	prototypage et vérification	prototypage et vérification
Limites	Abscence de temps		

III.1. Approche

La première distinction entre les réseaux prédicats/transitions et les langages que sont P.I.P.N. et L.O.T.O.S. repose sur l'approche, structurée ou non, de l'élaboration de la spécification.

Les réseaux prédicats/transitions nous permettent de spécifier une version simpliste et idéaliste de la tâche puis, par élaboration incrémentale, de s'étendre vers les spécifications complexes finales. En effet, comme nous l'avons évoqué au chapitre 2, une place peut, par exemple, être remplacée par un sous-réseau. Ce mécanisme de raisonnement est également présent dans le langage de spécifications L.O.T.O.S. mais s'applique alors au processus. En effet, un processus peut être décrit comme la composition séquentielle de deux sous-processus qui seront détaillés par la suite. Rappelons toutefois que le raffinement de la spécification élaborée en réseaux prédicats/transitions, reflet du niveau d'abstraction, s'effectue sur un seul et même schéma car l'absence d'opérateur ne permet pas de structurer la spécification, ce qui peut aboutir à des spécifications de taille prohibitive et dès lors rapidement illisibles.

Les langages P.I.P.N. et L.O.T.O.S. nous permettent, quant à eux, de spécifier un système par modules (respectivement processus) inter-reliés. Cette structuration de la spécification nous offre un degré de réutilisabilité et de maintenabilité plus élevé. En effet, un module (processus) peut être réutilisé dans d'autres spécifications. Bien entendu, une telle approche n'est possible que si les différents composants ont été parfaitement bien identifiés et leurs dépendances spécifiées en vue d'assurer la cohérence comportementale des composants une fois assemblés.

III.2. L'axe de description

Les réseaux prédicats/transitions sont composés de deux types d'objets : les états du système et les transitions. Traditionnellement, leur syntaxe, tout comme celle des réseaux de Pétri, est associée à l'approche dite d'ordre partiel lorsqu'on tente de définir les comportements du système considéré en terme d'exécution. L'approche dite d'ordre partiel considère que les événements sont reliés par deux relations : causalité et exclusion.

A ces deux relations s'ajoutent, dans le langage P.I.P.N., le principe d'unification des transitions. L'échange de messages entre deux modules inter-reliés se réalise par l'émission ('output') d'une donnée par le module émetteur et par la réception ('input') de cette même donnée par le module à qui le message est destiné. Par exemple, l'arbitre de bus envoie une requête d'identification sur le bus :

```
output (phy(id_dat(_ID)))
```

où phy est le port de communication, id_dat la requête et _ID l'instanciation d'une variable. La station reçoit cette requête. Si celle-ci s'unifie avec un message en entrée d'une transition, elle permettra le tir de cette transition à condition bien évidemment que ses préconditions soient vérifiées. Dans le module 'station', l'unification sera possible si nous avons :

```
input (phy(id_dat(_ID)))
```

L'accent est mis sur les transitions et leur unification. La notion d'état existe et transparait au travers des pré- et des postconditions.

En L.O.T.O.S, par contre, la notion d'état est rendue implicite. La description s'y axe sur les transitions et les types abstraits de données. Seules les actions observables permettent de saisir le comportement d'un processus. Ces actions peuvent faire intervenir des données dont l'utilisation dans des spécifications plus complexes s'avère un atout indispensable, surtout étant donné l'absence de structure de variables. Si, de la même façon que pour les réseaux prédicats/transitions et le langage P.I.P.N., nous essayons de définir les comportements du système en terme d'exécutions, nous constatons que cette approche est souvent associée à l'approche dite de branchement. Les exécutions sont regoupées dans un 'arbre de calcul', c'est-à-dire un système de transition qui ne reprend que les observations suivies d'exécutions divergeantes. Cette approche de branchement a l'inconvénient de ne pas distinguer la concurrence du non déterminisme.

III.3. La sémantique

L'indépendance des transitions et les réseaux de Pétri nous permettent d'interpréter n'importe quel réseau prédicats/transitions. En effet, ce dernier constitue une abstraction des réseaux de Pétri au sens où ils permettent de décrire le test puis le positionnement d'une condition par une seule transition atomique. Dans les réseaux de Pétri, cette condition est décomposée en une transition de début de test et une transition de fin de test. Ce mécanisme de décomposition n'assure en aucun cas l'indépendance des deux transitions, ce

qui signifie qu'elles n'ont aucune place connectée commune. Ce n'est pas le cas pour les réseaux prédicats/transitions.

Rappelons que ces réseaux peuvent être augmentés par la notion d'étiquetage. Celle-ci permet d'envisager un système comme un agglomérat de sous-réseaux interreliés, dont le mode de communication de base dans le langage P.I.P.N. est le rendez-vous. Dans ce cas, l'échange de message résulte de l'unification des paramètres des interactions complémentaires de deux transitions qui sont données par la clause 'connect'. Cette dernière déclaration peut être complétée par un attribut "fifo". Dans ce cas, l'échange de messages ne s'effectue plus de manière simultanée mais via une file "First In First Out". Chaque module de la déclaration 'connect (..., fifo)' est synchronisé par rendez-vous avec le module fifo. Ce mécanisme de rendez-vous est inspiré des langages L.O.T.O.S. et C.C.S En effet, il peut être déduit, de ce dernier, par les opérateurs de composition et de restriction.

Enfin, la sémantique de L.O.T.O.S. est essentiellement basée sur les systèmes de transitions entrelacées. Elle résulte de l'approche non plus orientée état, comme dans les deux cas précédents, mais de celle orientée "événement". Les séquences d'événements ou action, comme par exemple l'émission de la valeur de l'identificateur après la réception de ce dernier, sont directement prises en considération au détriment des états intermédiaires. De plus, cette sémantique est opérationnelle et liée à la théorie de l'algèbre libre de processus C.C.S., comme nous avons pu le constater lors du chapitre précédent.

III.4. Bases théoriques

La logique des prédicats du premier ordre est sous-jacente tant aux réseaux prédicats/transitions qu'au langage P.I.P.N. Elle permet notamment d'exprimer les conditions sur les données à l'aide de formules ou prédicats. Ces formules sont décrites sous forme de clauses de Horn qui déterminent la véracité d'une hypothèse à l'aide des ses propriétés. Il se peut toutefois que l'ensemble des propriétés soit vide ; nous sommes alors confrontés à certaines caractéristiques des données du problème, comme par exemple 'producteur(répondeur,other)' ; ce dernier est désigné sous le terme de prédicats statiques, c'est-à-dire dont la valeur de vérité est invariante. Si cette valeur est mise à jour avec l'état du système, comme pour les pré- et postconditions, nous parlerons de prédicats dynamiques. Notons que les clauses de Horn peuvent être résolues en Prolog ; ce langage est utilisé par P.I.P.N. notamment lors du dépliage d'une transition. Le dépliage nous permet d'obtenir le réseau de Pétri équivalent à notre modèle, lequel pourra de ce fait être analysé.

En outre, P.I.P.N. s'inspire du langage C.C.S. Celui-ci lui procure un moyen efficace de vérification par projection et un mode de communication aisé entre deux modules. La technique de projection permet d'obtenir le service offert par le protocole modélisé suivant un critère d'observation. Citons la projection sur un point d'interaction donné ou encore une projection de type "langage" [PIPN 88]. Ces projections nous permettent de mieux saisir le comportement de chaque module pris (ou non) séparément. En effet, la projection sur un point d'interaction consiste à n'envisager que les transitions dont le label comporte une interaction sur le port de communication envisagé et ce, par équivalence observationnelle. Rappelons que le mode de communication, également inspiré de C.C.S. , (et de L.O.T.O.S.) est le rendez-vous, sur lequel nous

reviendrons dans la sous-section suivante, consacrée aux mécanismes de synchronisation.

Le langage L.O.T.O.S. se base, pour sa part, non seulement sur cette algèbre de processus qu'est C.C.S. mais également sur le langage ACT-ONE. Ce dernier lui permet de prendre en considération des types de données complexes tels que des tables de routage. Cependant, il est parfois nécessaire de recourir à de nombreux types intermédiaires. Ainsi, une table de routage est une table qui comporte des entrées, des identificateurs de passerelle... Il faut dès lors renommer la table en un type intermédiaire, en vue de l'enrichir. Cette approche est progressive mais assez lourde et sans aucun dynamisme. Elle nous permet de compléter la partie de description de processus décrite à partir d'une version révisée de C.C.S. dans le cadre du projet S.E.D.O.S. et sur laquelle nous nous sommes penchée tout au long du chapitre 3. L'introduction de ces types abstraits de données permet d'adopter un niveau d'abstraction plus élevé dans cette dernière partie.

III.5. Les mécanismes de synchronisation

Les réseaux prédicats/transitions ne possèdent pas à proprement parler de mécanismes de synchronisation. Le tir d'une transition ne dépend que de la satisfaction des préconditions et des conditions de test puisqu'aucun mécanisme de structuration n'est mis en oeuvre, à l'inverse des langages P.I.P.N. et L.O.T.O.S. Dans ceux-ci, nous rencontrons un mécanisme commun de synchronisation (entre modules/processus), le rendez-vous, qui découle de la théorie commune sous-jacente qu'est C.C.S. Rappelons toutefois que si cet opérateur peut être déduit de cette algèbre de processus, il a été introduit en L.O.T.O.S. suite aux propositions faites dans le cadre des

améliorations de langage proposées dans le projet S.E.D.O.S. ; aussi, P.I.P.N. s'inspire t-il de L.O.T.O.S. Le rendez-vous permet à deux modules (ou processus) de communiquer lorsque simultanément l'un est prêt à recevoir l'interaction et l'autre à l'émettre.

En L.O.T.O.S. , cette notion peut être nuancée car elle englobe trois cas distincts. Dans le premier cas, la réception ne se distingue d'aucune façon de l'émission de l'interaction : les deux modules offrent la même valeur au même point d'interaction. Il s'agit simplement d'une synchronisation. Dans le deuxième cas, la réception se décrit à l'aide de variables de type donné auquel doit appartenir la valeur de l'interaction émise. La synchronisation peut être interprétée comme un passage de valeur. Enfin, dans le troisième cas, l'émission et la réception se notent par l'attente d'une valeur qui s'identifie à une variable d'un type donné et qui est inconnue avant l'événement. L'interprétation est plutôt la génération d'une valeur quelconque du type considéré. Ces trois cas permettent cependant une communication simultanée entre les différents processus, ce qui n'est pas le cas si nous envisageons le second mode de communication offert par le langage P.I.P.N. Dans celui-ci, deux modules peuvent communiquer par l'intermédiaire d'une file "First In First Out". La communication entre deux processus n'est donc plus concomitante.

Les mécanismes de synchronisation reflètent le parallélisme et la concurrence entre deux modules (ou processus). En effet, les modules interreliés peuvent évoluer de manière autonome jusqu'à une offre d'événement. Leur interconnexion se réalise alors en P.I.P.N., par la clause 'connect', et en L.O.T.O.S. par l'opérateur de parallélisme avec interactions. Une faiblesse de ces langages réside toutefois dans

l'impossibilité immédiate de décrire la mutuelle exclusion ou synchronisation par section critique. En L.O.T.O.S., il s'avère nécessaire d'introduire un processus intermédiaire et de redéfinir les processus pris en considération. Un subterfuge similaire est utilisé dans le langage P.I.P.N. afin de pallier la contrainte d'indépendance.

III.6. Techniques d'analyse

Nous ne nous attarderons pas sur les différents aspects d'analyse liés à ces trois formalismes. Ces aspects ne sont cités qu'à titre indicatif et sont amplement détaillé dans la littérature.

Sous le terme de 'techniques d'analyse', nous englobons les aspects de vérification et de validation de la spécification formelle élaborée. Rappelons que la vérification s'attache aux aspects syntaxiques et sémantiques propres au formalisme ainsi qu'à l'obtention de l'automate ; tandis que la validation se préoccupe davantage de l'analyse et de la simulation de l'automate obtenu.

Les réseaux prédicats/transitions utilisent les mêmes techniques d'analyse que celles utilisées par les réseaux de Pétri à savoir les invariants et le graphe des marquages [Peterson 81]. C'est sur l'utilisation de ce dernier que repose toute vérification (et analyse) dans le langage P.I.P.N. Celui-ci nous offre trois techniques : la projection, la logique temporelle et les propriétés spécifiques. La projection permet, dans le cas qui nous occupe, d'obtenir les modèles du service et du médium et de détecter les blocages internes. La logique temporelle teste l'accessibilité d'un état à l'aide de formules. Les propriétés spécifiques mettent en évidence les cycles de transitions et les états de blocage [Lloret 88].

Le langage L.O.T.O.S. ne nous permet pour l'instant que de vérifier l'équivalence de comportement grâce à C.C.S., sa base mathématique, bien que d'autres possibilités de vérification soient actuellement en cours d'étude, comme nous l'avons souligné au chapitre précédent.

III.7. Types d'application

Les réseaux prédicats/transitions sont souvent utilisés dans une perspective de modélisation et d'analyse de systèmes dynamiques. En effet, la complexité croissante des technologies d'automatisation rend difficile des analyses et des évaluations fondées uniquement sur le savoir-faire et l'expérience accumulés. Les systèmes de production doivent être de plus en plus flexibles. Ils remettent en cause les structures séquentielles et engendrent par la même occasion de nouveaux problèmes de coordination des activités parallèles. Leur modélisation permet non seulement de saisir en cours d'étude quelles sont les caractéristiques essentielles du système mais également de mettre en évidence de nouvelles connaissances sans aucun danger [Pe 81]. Remarquons que le domaine d'application de ces réseaux, et par la même occasion celui des réseaux de Pétri, s'élargit chaque jour davantage pour diverses raisons. Citons entre autres leurs facilités d'utilisation, leurs constructions progressives, leurs propriétés communes avec les protocoles de communication et enfin les nombreux efforts développés pour construire des outils d'édition et de validation. Ainsi, le L.A.A.S. a-t-il développé l'outil P.I.P.N. Celui-ci permet d'éditer, d'analyser et de simuler le modèle et constitue un excellent outil de prototypage rapide. Un regret reste à formuler : son utilisation restreinte essentiellement à son milieu de

développement et la quasi impossibilité de prendre en compte le traitement des données.

Le langage L.O.T.O.S. offre un pouvoir d'expression plus adéquat pour la spécification de systèmes distribués complexes. Tout comme l'outil P.I.P.N., les outils qui le supportent permettent l'édition, le prototypage et la vérification de la spécification.

III.8. Limites

Les trois modèles que nous avons envisagés présentent tous la même lacune : le temps. Celle-ci nous semble importante puisque de manière générale, les systèmes distribués sont soumis à des contraintes temporelles comme par exemple, le temps de réponse. Certaines extensions des formalismes qui ont été présentés dans ce mémoire tiennent spécialement compte de cette notion fondamentale : les réseaux de Pétri temporels (et temporisés) et "Timed L.O.T.O.S."

IV. Perspective et choix

Ces trois formalismes ne représentent qu'un sous-ensemble restreint des techniques de description formelle. Celles-ci comprennent notamment les langages Estelle et S.D.L. et les machines à états finies [Courtiat 87]. Cette multiplicité de techniques engendre des difficultés dans le choix d'un formalisme adéquat pour la spécification et la validation d'un système. En effet, ce choix n'est pas uniquement guidé par les

caractéristiques théoriques que nous venons d'évoquer ; l'expérience, les moyens disponibles, l'objectif poursuivi seront quelques uns des autres facteurs pris en considération. Nous allons exposer brièvement quelques considérations d'ordre pratique relatives aux formalismes que nous avons utilisés dans le cadre de l'élaboration de la spécification de la couche liaison de données de Fip.

Le travail effectué permet de mettre en évidence l'importance fondamentale accordée aux exemples disponibles. En effet, les exemples facilitent la compréhension des mécanismes parfois complexes de synchronisation mis en oeuvre tant dans L.O.T.O.S. que dans P.I.P.N. et nous permettent de nous familiariser avec ces langages. Les connaissances du concepteur influencent cet apprentissage. Ainsi le fait que nous connaissions les réseaux de Pétri nous a grandement aidé lors de l'étude des réseaux prédicats/transitions et du langage P.I.P.N. Ces deux derniers langages nous apparaissent toutefois peu adéquats pour la spécification d'un système. Ils introduisent tous deux des définitions 'immatérielles', telle que la notion d'état, et ne se limitent pas à la description du comportement observable externe. En conséquence, ces techniques entraînent non seulement une surspécification mais restreignent la liberté du concepteur qui se perd souvent dans les dédales du processus de spécification. Si, dorénavant, nous envisageons comme objectif, non plus la spécification, mais le prototypage rapide, c'est l'outil P.I.P.N. qui s'avère le mieux adapté. En effet, il ne fait pas intervenir la notion de types de données, ce qui augmente la rapidité du prototypage. De plus, la réalisation d'une offre d'événement, par exemple la réception par le 'medium' de la requête 'Iddat(_Id)' , qui donne lieu a une synchronisation des différentes entités qui y sont connectées, permet d'obtenir, lors de la simulation du prototype, une vision complète de l'état des différents

modules impliqués par l'offre d'événement. Le choix de la transition est alors laissée à la discrétion de l'utilisateur. En L.O.T.O.S. , l'utilisateur perçoit la synchronisation des modules deux par deux et de manière successive. Il ne dispose donc pas d'une vision globale de l'impact de la réalisation de l'événement. De plus, la personne qui élabore le prototype doit décrire les types de données auxquels font références les offres d'événements, lesquelles peuvent être relativement complexes. Aussi, si chaque langage a une utilité spécifique :

Les réseaux prédicats/transitions : la modélisation,
P.I.P.N. : le prototypage rapide
L.O.T.O.S. : la spécification

nous devons nous rendre à l'évidence qu'ils ne seront pas automatiquement utilisés à cette fin. Des éléments autres que ceux évoqués précédemment entrent en ligne de compte tels que les moyens disponibles et l'optique dans laquelle le formalisme est choisi : l'apprentissage personnel et/ou professionnel ou l'enseignement Selon nous, les réseaux de Pétri et ses dérivés constituent un excellent outil pédagogique. Ils servent de référence sémantique à de nombreux autres langages ; leurs techniques d'analyse permettent une prise de conscience de leur pouvoir de modélisation et leur support graphique représente un attrait non négligeable pour les étudiants. De plus, de multiples cas d'école très simples peuvent être utilisés et expliqués. Il n'en va pas de même pour des langages tels que P.I.P.N. ou L.O.T.O.S. dont les mécanismes de structuration, pour être bien compris, nécessitent des exemples un peu plus complexes. Toutefois, un cas d'école décrit dans le modèle des réseaux de Pétri (ou l'un de ses dérivés) peut facilement être transcrit, s'il est suffisament raffiné, dans les langages que sont

P.I.P.N. et L.O.T.O.S. Quelques règles systématiques sont applicables aux réseaux prédicats/transitions pour l'obtention du modèle P.I.P.N. équivalent. Pour notre part, nous avons étudié, sur base du système Fip et du système téléphonique, les principes de base qui nous permettent de passer indifféremment du modèle P.I.P.N. à la spécification L.O.T.O.S. Une étude syntaxique procure la charpente de la spécification ; les conditions deviennent des 'gardes', les 'inputs' des '?', ... et inversément. Le problème majeur rencontré est évidement la définition des types abstraits. Une étude approfondie de tels principes et de leur implémentation offriraient un moyen aisé pour élaborer un système à l'aide d'outils couvrant tout le cycle de conception. Une traduction entièrement automatique nous semble cependant quelque peu utopique bien que nous songions à la réalisation d'un outil d'aide à la transformation interactive de ces spécifications. Un tel travail mériterait un mémoire à lui seul et les quelques éléments que nous avons retirés ne sont donnés qu'à titre exemplatif.

Retenons essentiellement que la connaissance du formalisme de base des réseaux de Pétri (ou dérivés) est un atout non négligeable pour la compréhension des langages P.I.P.N. et L.O.T.O.S. mais elle n'est pas indispensable. Si une personne non avertie désire apprendre un de ces deux langages, nous lui conseillerons de commencer par le langage L.O.T.O.S. Celui-ci permet de décrire les services attendus par le système, d'une manière proche du langage naturel c'est-à-dire par son comportement observable externe. Son apprentissage est également facilité par la connaissance des langages procéduraux. Bien sûr, son acquisition est quelque peu complexe : manipulation des types abstraits de données et description de processus à l'aide d'une syntaxe pas toujours évidente. Mais il permet de bien comprendre le comportement global (et local) du système en ignorant des détails tels que

les états qui nous semblent superflus et souvent difficiles à choisir correctement. Ces états sont facilement déduisibles de la spécification, de par la nature même de L.O.T.O.S. Non seulement son degré d'abstraction est-il plus adéquat mais son apprentissage se révèle très enrichissant : étude d'une algèbre de processus et d'un langage de types abstraits. Enfin, il constitue désormais un langage normalisé alors que l'utilisation de P.I.P.N. est aujourd'hui encore fort limitée. Les outils environnementaux de L.O.T.O.S. se multiplient. Ses enseignements apportent leur contribution à divers projets. Cette prise en considération de L.O.T.O.S. et les travaux dont il est le sujet permettent d'espérer qu'il sera d'ici peu aussi capable que P.I.P.N. d'effectuer les mêmes vérifications.

V. Conclusion

Dans ce chapitre, nous avons esquisser quelques unes des différences entre trois formalismes : les réseaux prédicats/transitions, P.I.P.N. et L.O.T.O.S. Notre approche s'est essentiellement basée sur les constatations issues de leur utilisation et les aspects théoriques que cette dernière a mis en relief. Bien que notre étude soit d'une portée très limitée, elle nous permet de percevoir l'adéquation de L.O.T.O.S. pour la spécification de protocoles de communication et la progression des formalismes utilisés : les concepts sont repris et enrichis. Citons la logique des prédicats du premier ordre, sous-jacente aux réseaux prédicats/transitions, à laquelle s'ajoute l'algèbre de processus C.C.S. dans P.I.P.N. ou encore la version révisée de C.C.S. à laquelle s'adjoint le langage ACT ONE dans L.O.T.O.S. Nous pouvons également remarquer que les systèmes de transitions (et réseaux de Pétri) constitue la clé de voûte de ces divers formalisme bien que leur pouvoir d'expression soit

assez restreint. Enfin, nous suggérons quelques conseils d'utilisation aux lecteurs non encore initiés à ces langages : l'apprentissage du langage L.O.T.O.S. dans une optique de spécification (et même de prototypage) de systèmes répartis, l'équivalent P.I.P.N. (ou réseaux prédicats/transitions) en étant facilement déductible.

Conclusion générale

Dans ce mémoire, nous nous sommes basée sur nos travaux relatifs à la spécification de la couche liaison de données de Fip en L.O.T.O.S. pour présenter une synthèse de quelques formalismes supports pour la spécification de protocole : les réseaux de Pétri (et quelques dérivés), P.I.P.N. et L.O.T.O.S. Rappelons les principales contributions apportées avant de conclure sur les extensions possibles.

Le chapitre premier décrit le protocole Fip en insistant particulièrement sur la couche liaison de données. L'approche choisie permet d'aborder deux styles de spécification : la spécification en langage naturel, issue de la description qui nous a été fournie au LAAS et une spécification graphique, plus personnelle. Ces différents styles de spécification nous permettent d'introduire la nécessité de disposer de "techniques de description formelle" afin de pallier leur ambiguïté et la dépendance du langage naturel pour exprimer les concepts techniques. Actuellement, nous trouvons parmi ces techniques, les réseaux de Pétri et ses dérivés, l'approche algébrique C.C.S., les types abstraits algébriques, L.O.T.O.S. ou encore P.I.P.N.

Dans le chapitre II, nous avons considéré les réseaux de Pétri comme modèle de base et nous avons développé les extensions qui nous permettaient d'aboutir de manière naturelle au langage P.I.P.N. En effet, celui-ci est une extension des réseaux prédicats/transitions, eux-mêmes extensions des réseaux de Pétri. Nous nous sommes familiarisée à ces différents formalismes par leur application au système téléphonique et par leur étude sur le protocole Fip. Notons que l'application précédente n'a été que citée dans le cadre de ce mémoire même si elle reste disponible.

Dans le chapitre III, nous avons introduit l'approche algébrique C.C.S. dont l'intérêt est triple : premièrement, elle permet de décrire et de structurer les systèmes de transitions. Deuxièmement, elle rend possible une extension aisée de l'algèbre par l'ajout d'un constructeur. Enfin, elle offre une axiomatisation de l'équivalence observationnelle. Cette équivalence permet, faut-il le rappeler, de déterminer si aucun événement ne distingue deux processus. Toutefois, cette approche comporte deux inconvénients majeurs : d'une part, la non prise en considération du traitement des données et d'autre part, l'impossibilité d'une synchronisation multiple (deux processus au plus peuvent communiquer ensembles). Ceux-ci ont valu une révision de C.C.S., dans le cadre du projet S.E.D.O.S., et l'introduction du langage ACT ONE en vue d'aboutir à un langage de spécification des services et protocoles du modèle OSI : le langage L.O.T.O.S. Nous avons introduit ses concepts à l'aide de notre spécification dont le procédé d'élaboration a été brièvement énoncé, pour finalement exposer la partie du projet S.E.D.O.S. relative à ce langage. En effet, une autre partie a été consacrée à un autre langage de spécification : Estelle. Notons que lors de l'introduction des concepts, nous n'avons pas évoqué une traduction personnelle des opérateurs L.O.T.O.S. en réseaux de Pétri (également disponible) ; elle nous a permis de mieux comprendre leur signification puisque nous connaissions le formalisme cible, et ce, pour des raisons d'ordre purement pratique.

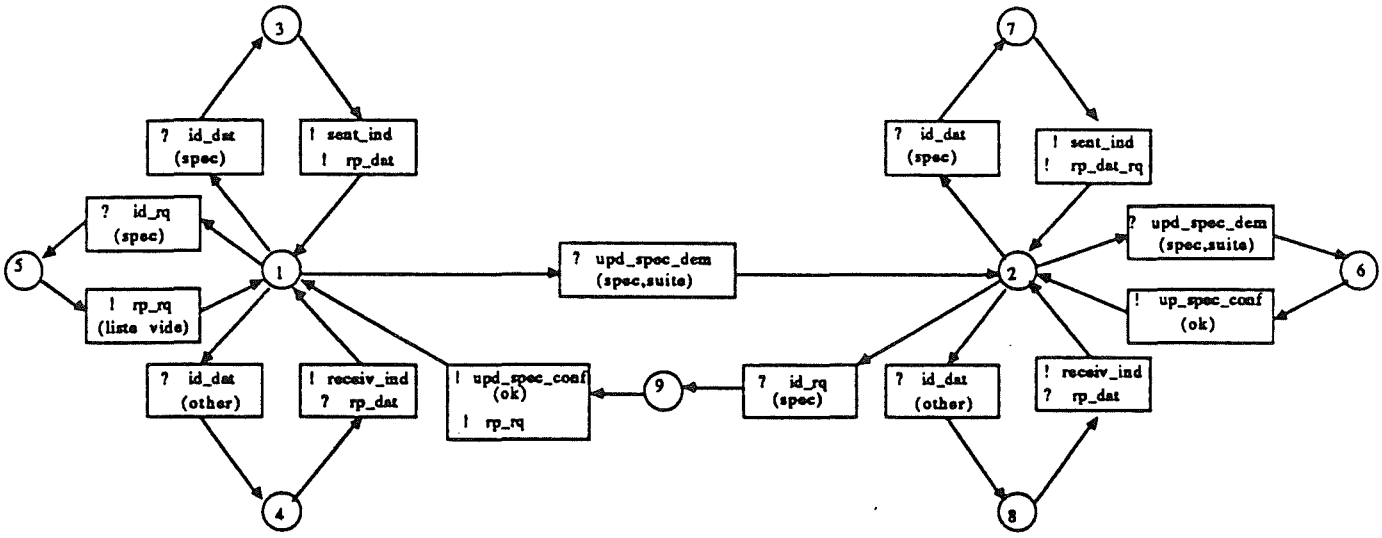
Finalement, dans le quatrième chapitre, nous avons ébauché une comparaison des trois formalismes avec lesquels nous avons travaillé : les réseaux de Pétri, P.I.P.N. et L.O.T.O.S. Cette comparaison repose aussi bien sur les aspects théoriques soulevés lors des travaux que sur leur utilisation respective. Elle nous explique les différences de niveaux auxquels se situe chaque formalisme lors de la

conception d'un protocole ; lors de la comparaison, elle met en évidence le choix des réseaux prédicats/transitions en lieu et place des réseaux de Pétri. Elle est très succincte puisqu'elle repose sur des constatations issues de nos applications et une connaissance d'ensemble des formalismes envisagés. Elle se veut exemplative et source de réflexion. En effet, nous ne nous sommes pas intéressée aux aspects mathématiques de validation et de vérification de ces différents formalismes mais nous nous sommes limitée à quelques observations liées à la spécification d'un protocole de communication, domaine trop restreint pour une étude complète et approfondie.

Les divers travaux effectués nous ont permis de bien percevoir les concepts de L.O.T.O.S. Ils ne nous ont pas donné l'occasion de pousser plus avant notre réflexion sur l'étude comparative et plus particulièrement sur les conseils d'utilisation liés aux Réseaux de Pétri et aux langages P.I.P.N. et L.O.T.O.S. Nous espérons que les différents travaux évoqués, plus particulièrement l'étude comparative, centre d'intérêt de ce mémoire, auront su mettre en relief les différents éléments susceptibles de générer un examen plus approfondi.

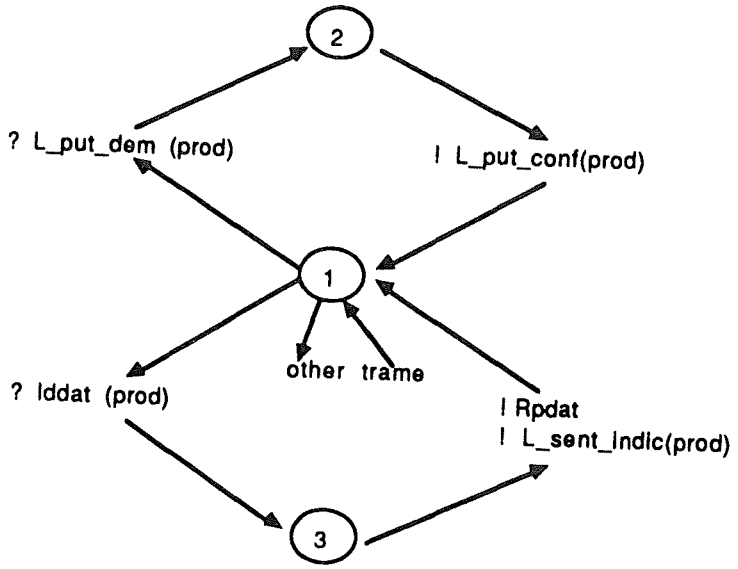
Annexes A : Le modèle des réseaux de Pétri élaboré par Nordgard

- Modèle général du service périodique de la couche liaison de données de Fip

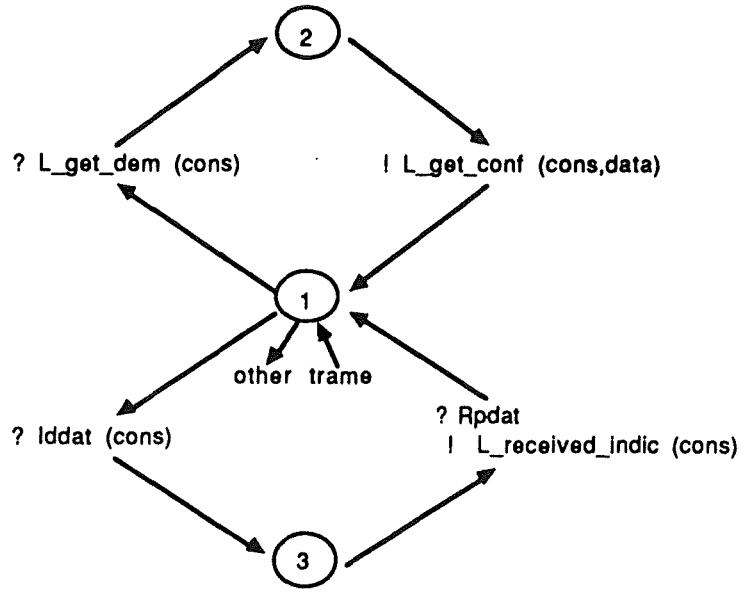


model : spec
primitive : update_spec_demande
identificateur : spec, other
automate : protocole

- Modèle producteur



- Modèle consommateur



Annexes B : Modèles P.I.P.N. des services périodiques et apériodiques spécifiés de la couche liaison de données de Fip

```
(* ========================================================================  *)
(* PIPN : SERVICE PERIODIQUE DE LA COUCHE LIAISON DE DONNEES                 *)
(* ========================================================================  *)


(* ---------------- *)
(* FIP.header.pro   *)
(* ---------------- *)


{ interface for fip }


entity(fip).

child([medium,station,repondeur,arbitre]).

connect(station(output(phy(_X))),medium(input(phys(_X)))).
connect(repondeur(output(phy(_X))),medium(input(phyr(_X)))).


connect(medium(output(phys(_X))),statin(input(phy(_X)))).
connect(medium(output(phyr(_X))),repondeur(input(phy(_X)))).


connect(medium(output(phy(_X))),arbitre(input(phy(_X)))).
connect(arbitre(output(phy(_X))),medium(input(phy(_X)))).


(* ------------------------------------------------------------------------ *)

(* LE MEDIUM *)


(* ----------------- *)
(* Medium.header.pro *)
(* ----------------- *)


entity(medium).

body(medium).

ip([input(phy(_)),
    output(phy(_)),
    input(phys(_)),
    output(phys(_)),
    input(phyr(_)),
    output(phyr(_))]).


(* --------------- *)
(* Medium.body.pro *)
(* --------------- *)


{ body for medium}

place([idle]).

init([idle]).

tr(rec_arbitre,
   label([input(phy(_X)),
          output(phyr(_X)),
          output(phys(_X))]),
   pre([idle]),
   post([idle])).

tr(rec_station,
   label([input(phys(_X)),
          output(phyr(_X)),
          output(phy(_X))]),
   pre([idle]),
   post([idle])).

tr(rec_repondeur,
   label([input(phyr(_X)),
          output(phys(_X)),
          output(phy(_X))]),
   pre([idle]),
   post([idle])).


(* ------------------------------------------------------------------------  *)

(* LA STATION *)


(* ------------------ *)
(* Station.header.pro *)
(* ------------------ *)

{ interface for station globale }

entity(station).

cond(station).

body(station).

ip([input(l(_)),
    input(phy(_)),
    output(l(_)),
    output(phy(_))]).

init([ idle, name(station), b_dat_prod(prod,dat_p), b_dat_cons(cons,dat_c) ]).




(* ---------------- *)
(* Station.body.pro *)
(* ---------------- *)


{body prod}

{TRANSITIONS POUR INTERFACE AVEC LA COUCHE APPLICATION}

{.... Service "put"}


tr(prepare_to_put,
   label([input(l(l_put_dem(_ID,_VAL1)))]),
   pre([idle,b_dat_prod(_ID,_VAL)]),
   post([put(_ID),b_dat_prod(_ID,_VAL1)]),
```

```
   cond([get_val(_ID,_VAL1)])).


tr(put_data,
   label([output(l(l_put_conf(_ID)))]),
   pre([put(_ID)]),
   post([idle])).



{.... Service "get"}

tr(prepare_to_get,
   label([input(l(l_get_dem(_ID)))]),
   pre([idle,b_dat_cons(_ID,_VAL)]),
   post([get(_ID,_VAL),b_dat_cons(_ID,_VAL)])).


tr(get_data,
   label([output(l(l_get_conf(_ID,_VAL)))]),
   pre([get(_ID,_VAL)]),
   post([idle])).



{TRANSITIONS POUR ECHANGES AVEC LA COUCHE PHYSIQUE}

tr(prepare_to_send,
   label([input(phy(id_dat(_ID)))]),
   pre([idle,name(_ME)]),
   post([send(_ME,_ID),name(_ME)]),
   cond([producteur(_ME,_ID)])).


 tr(send_rp_dat,
   label([output(phy(rp_dat(_VAL))),
          output(l(l_sent_ind(_ID,_VAL)))]),
   pre([send(_ME,_ID),b_dat_prod(_ID,_VAL)]),
   post([idle,b_dat_prod(_ID,_VAL)]),
   cond([id_status(_ME,_ID,per)])).


tr(prepare_to_read,
   label([input(phy(id_dat(_ID)))]),
   pre([idle,name(_ME)]),
   post([read(_ID),name(_ME)]),
   cond([consommateur(_ME,_ID)])).


tr(read_rp_dat,
   label([input(phy(rp_dat(_VAL))),
          output(l(l_received_ind(_ID)))]),
   pre([read(_ID),b_dat_cons(_ID,_)]),
   post([idle,b_dat_cons(_ID,_VAL)])).


(* ------------------------------------------------------------------------ *)

(* LE REPONDEUR *)



(* -------------------- *)
```

```
(* Repondeur.header.pro *)
(* -------------------- *)


{ interface for station globale }

entity(repondeur).

body(repondeur).

ip([input(l(_)),
    input(phy(_)),
    output(l(_)),
    output(phy(_))]).


init([ idle, name(repondeur), b_dat_prod(cons,dat_c), b_dat_cons(prod,dat_p) ]).




(* ------------------ *)
(* Repondeur.body.pro *)
(* ------------------ *)


{TRANSITIONS POUR ECHANGES AVEC LA COUCHE PHYSIQUE}

tr(prepare_to_send,
   label([input(phy(id_dat(_ID)))]),
   pre([idle,name(_ME)]),
   post([send(_ME,_ID),name(_ME)]),
   cond([producteur(_ME,_ID)])).


 tr(send_rp_dat,
   label([output(phy(rp_dat(_VAL))),
          output(l(l_sent_ind(_ID,_VAL)))]),
   pre([send(_ME,_ID),b_dat_prod(_ID,_VAL)]),
   post([idle,b_dat_prod(_ID,_VAL)]),
   cond([id_status(_ME,_ID,per)])).


tr(prepare_to_read,
   label([input(phy(id_dat(_ID)))]),
   pre([idle,name(_ME)]),
   post([read(_ID),name(_ME)]),
   cond([consommateur(_ME,_ID)])).


tr(read_rp_dat,
   label([input(phy(rp_dat(_VAL))),
          output(l(l_received_ind(_ID)))]),
   pre([read(_ID),b_dat_cons(_ID,_)]),
   post([idle,b_dat_cons(_ID,_VAL)])).


(* ------------------------------------------------------------------------ *)

(* L'ARBITRE  *)


(* ------------------ *)
(* Arbitre.header.pro *)
```

```
(* ------------------ *)

{ interface for arbitre }


entity(arbitre).

cond(arbitre).

body(arbitre).

ip([input(phy(_)),
    output(phy(_))]).

init([next(per), id_per([prod,cons])]).



(* ---------------- *)
(* Arbitre.body.pro *)
(* ---------------- *)


tr(send_id_dat_per,
   label([output(phy(id_dat(_ID)))]),
   pre([next(per),id_per([_ID|_T])]),
   post([wait_rp_dat(_ID,per),id_per(_T)]),
  ).

tr(receive_rp_dat,
   label([input(phy(rp_dat(_)))]),
   pre([wait_rp_dat(_,_S)]),
   post([next(_S)])).



(* ---------------- *)
(* Arbitre.cond.pro *)
(* ---------------- *)

id_per_fix([prod,cons]).
```

```
(* ========================================================================  *)
(* PIPN :  SERVICE APERIODIQUE SPECIFIE DE LA COUCHE LIAISON DE DONNEES  *)
(* ========================================================================  *)



(* -------------- *)
(* FIP.header.pro *)
(* -------------- *)


{ interface for fip }


entity(fip).

child([medium,station,repondeur,arbitre]).

connect(station(output(phy(_X))),medium(input(phys(_X)))).
connect(repondeur(output(phy(_X))),medium(input(phyr(_X)))).


connect(medium(output(phys(_X))),station(input(phy(_X)))).
connect(medium(output(phyr(_X))),repondeur(input(phy(_X)))).


connect(medium(output(phy(_X))),arbitre(input(phy(_X)))).
connect(arbitre(output(phy(_X))),medium(input(phy(_X)))).


(* ------------------------------------------------------------------------  *)

(* LE MEDIUM *)


(* ----------------- *)
(* Medium.header.pro *)
(* ----------------- *)

entity(medium).

body(medium).

ip([input(phy(_)),
    output(phy(_)),
    input(phys(_)),
    output(phys(_)),
    input(phyr(_)),
    output(phyr(_))]).



(* --------------- *)
(* Medium.body.pro *)
(* --------------- *)

{ body for medium}

place([idle]).

init([idle]).

tr(rec_arbitre,
   label([input(phy(_X)),
          output(phyr(_X)),
          output(phys(_X))]),
   pre([idle]),
   post([idle])).

tr(rec_station,
   label([input(phys(_X)),
          output(phyr(_X)),
          output(phy(_X))]),
   pre([idle]),
   post([idle])).

tr(rec_repondeur,
   label([input(phyr(_X)),
          output(phys(_X)),
          output(phy(_X))]),
   pre([idle]),
   post([idle])).


(* ------------------------------------------------------------------------  *)


(* LA STATION *)



(* ------------------ *)
(* Station.header.pro *)
(* ------------------ *)

{ interface for station globale }

entity(station).

cond(aper).

body(station).

ip([input(l(_)),
    input(lm(_)),
    input(phy(_)),
    output(l(_)),
    output(lm(_)),
    output(phy(_))]).


init([ idle, name(aper_spec), b_dat_prod(spec,data), b_dat_cons(other,data),
       b_req(spec,[],0) ]).



(* ---------------- *)
(* Station.body.pro *)
(* ---------------- *)


place([idle,
       name(_ME),

       {Places pour service periodique:}

       b_dat_prod(_ID,_VAL),
```

```
        b_dat_cons(_ID,_VAL),
        get(_ID,_VAL),
        put(_ID),
        send(_ME,_ID),
        read(_ID),

        {Places pour service  "update_spec":}


        update_ecrase(_ID),
        update_unknown(_ID),
        b_req(_ID,_L,_RQ),
        send_aper_spec(_ID)]).


{TRANSITIONS POUR INTERFACE AVEC LA COUCHE APPLICATION}


{....  Service "update_spec":}


tr(receive_update_ok,
   label([input(l(l_update_spec_dem(_ID,_SERIE)))]),
   pre([idle,b_req(_ID,_,0)]),
   post([idle,b_req(_ID,_SERIE,1)]),
   cond([ get_serie(_SERIE) ])).

tr(receive_update_ecrase,
   label([input(l(l_update_spec_dem(_ID,_SERIE)))]),
   pre([idle,b_req(_ID,_,1)]),
   post([b_req(_ID,_SERIE,1),update_ecrase(_ID)]),
   cond([ get_serie(_SERIE) ])).


tr(send_update_ecrase,
   label([output(l(l_update_spec_conf(_ID,ecrase)))]),
   pre([update_ecrase(_ID)]),
   post([idle])).


{TRANSITIONS POUR ECHANGES AVEC LA COUCHE PHYSIQUE}



{.... Transitions pour une station productrice/consommatrice de donnees
 periodiques}

tr(prepare_to_send,
   label([input(phy(id_dat(_ID)))]),
   pre([idle,name(_ME)]),
   post([send(_ME,_ID),name(_ME)]),
   cond([producteur(_ME,_ID)])).


 tr(send_rp_dat,
   label([output(phy(rp_dat(_VAL))),
          output(l(l_sent_ind(_ID,_VAL)))]),
   pre([send(_ME,_ID),b_dat_prod(_ID,_VAL)]),
   post([idle,b_dat_prod(_ID,_VAL)]),
   cond([id_status(_ME,_ID,per)])).


tr(prepare_to_read,
   label([input(phy(id_dat(_ID)))]),
   pre([idle,name(_ME)]),
   post([read(_ID),name(_ME)]),
   cond([consommateur(_ME,_ID)])).


tr(read_rp_dat,
   label([input(phy(rp_dat(_VAL))),
          output(l(l_received_ind(_ID)))]),
   pre([read(_ID),b_dat_cons(_ID,_)]),
   post([idle,b_dat_cons(_ID,_VAL)])).



{.... Transitions pour une station configuree pour le service "update_spec"}


tr(send_rp_dat(spec),
   label([output(phy(rp_dat(_VAL))),
          output(l(l_sent_ind(_ID,_VAL)))]),
   pre([send(_ME,_ID),b_dat_prod(_ID,_VAL),b_req(_ID,_SERIE,0)]),
   post([idle,b_dat_prod(_ID,_VAL),b_req(_ID,_SERIE,0)])).


tr(send_rp_dat_rq(spec),
   label([output(phy(rp_dat_rq(_VAL))),
          output(l(l_sent_ind(_ID,_VAL)))]),
   pre([send(_ME,_ID),b_dat_prod(_ID,_VAL),b_req(_ID,_SERIE,1)]),
   post([idle,b_dat_prod(_ID,_VAL),b_req(_ID,_SERIE,1)]),
   cond([id_status(_ME,_ID,aper_spec_rq)])
   ).


tr(prepare_send_aper_spec,
   label([input(phy(id_rq(_ID)))]),
   pre([idle,b_req(_ID,_SERIE,_RQ)]),
   post([send_aper_spec(_ID),b_req(_ID,_SERIE,_RQ)])).


tr(send_rp_rq_bug,
   label([output(phy(rp_rq([])))]),
   pre([send_aper_spec(_ID),b_req(_ID,_SERIE,0)]),
   post([idle,b_req(_ID,_SERIE,0)])).


tr(send_rp_rq_spec,
   label([output(phy(rp_rq(_SERIE))),
          output(l(l_update_spec_conf(_ID,ok)))]),
   pre([send_aper_spec(_ID),b_req(_ID,_SERIE,1)]),
   post([idle,b_req(_ID,_SERIE,0)])).




(* ------------------------------------------------------------------  *)


(* LE REPONDEUR *)



(* -------------------- *)
(* Repondeur.header.pro *)
(* -------------------- *)
```

```
{ interface for station repondeur }

entity(repondeur).
cond(aper).
body(repondeur).

ip([input(l(_)),
    input(lm(_)),
    input(phy(_)),
    output(l(_)),
    output(lm(_)),
    output(phy(_))]).


init([ idle, name(repondeur), b_dat_prod(other,data)]).



(* ------------------ *)
(* Repondeur.body.pro *)
(* ------------------ *)



place([idle,
       name(_ME),


       {Places pour service periodique:}


       b_dat_prod(_ID,_VAL),
       b_dat_cons(_ID,_VAL),
       get(_ID,_VAL),
       put(_ID),
       send(_ME,_ID),
       read(_ID),

       {Places pour service  "update_spec":}


       update_ecrase(_ID),
       update_unknown(_ID),
       b_req(_ID,_L,_RQ),
       send_aper_spec(_ID)]).




{TRANSITIONS POUR ECHANGES AVEC LA COUCHE PHYSIQUE}



tr(prepare_to_send,
   label([input(phy(id_dat(_ID)))]),
   pre([idle,name(_ME)]),
   post([send(_ME,_ID),name(_ME)]),
   cond([producteur(_ME,_ID)])).


 tr(send_rp_dat,
   label([output(phy(rp_dat(_VAL))),
          output(l(l_sent_ind(_ID,_VAL)))]),
   pre([send(_ME,_ID),b_dat_prod(_ID,_VAL)]),
   post([idle,b_dat_prod(_ID,_VAL)]),
   cond([id_status(_ME,_ID,per)])).


tr(prepare_to_read,
   label([input(phy(id_dat(_ID)))]),
   pre([idle,name(_ME)]),
   post([read(_ID),name(_ME)]),
   cond([consommateur(_ME,_ID)])).


tr(read_rp_dat,
   label([input(phy(rp_dat(_VAL))),
          output(l(l_received_ind(_ID)))]),
   pre([read(_ID),b_dat_cons(_ID,_)]),
   post([idle,b_dat_cons(_ID,_VAL)])).




(* ------------------------------------------------------------------------  *)

(* Fichier de condition aper.cond.pro (associe a la station et au repondeur *)



producteur(aper_spec,spec).
producteur(repondeur,other).
consommateur(aper_spec,other).

id_status(aper_spec,spec,aper_spec_rq).
id_status(repondeur,other,per)


get_serie([other]).

not_aper_spec_id(_ME,_ID):- id_status(_ME,_ID,per).

no_rq_spec_cond(0,_,_):-!.
no_rq_spec_cond(1,_ME,_ID):- \+ id_status(_ME,_ID,aper_spec_rq).


(* ------------------------------------------------------------------------  *)


(* L' ARBITRE *)



(* ------------------ *)
(* Arbitre.header.pro *)
(* ------------------ *)

{ interface for arbitre }


entity(arbitre).

cond(arbitre).

body(arbitre).
```

```
ip([input(phy(_)),
    output(phy(_))]).

init([next(per), id_per([other,spec]), f_idrq([]), id_aper([]), aper_count(0)]).



(* ---------------- *)
(* Arbitre.body.pro *)
(* ---------------- *)


{ body for arbitre }


place([next(_S1),
       wait_rp_dat(_ID,_S2),
       wait_rp_rq(_S),
       id_per(_L),
       f_idrq(_L),
       aper_count(_N),



{transitions "switches" qui changent le mode d'operation de l'arbitre}



tr(switch_to_aper,
   pre([next(per),id_per([])]),
   post([next(aper),id_per([])])).



{PERIODIQUE.........}

tr(send_id_dat_per,
   label([output(phy(id_dat(_ID)))]),
   pre([next(per),id_per([_ID|_T])]),
   post([wait_rp_dat(_ID,per),id_per(_T)]),
   cond([ \+( _ID = idrq(_) )])).

tr(receive_rp_dat,
   label([input(phy(rp_dat(_)))]),
   pre([wait_rp_dat(_,_S)]),
   post([next(_S)])).

tr(receive_rp_dat_rq,
   label([input(phy(rp_dat_rq(_)))]),
   pre([wait_rp_dat(_ID,_S),f_idrq(_L)]),
   post([next(_S),f_idrq(_M)]),
   cond([update_idrq(_L,_M,_ID)])).

tr(send_id_rq_per,
   label([output(phy(id_rq(_ID)))]),
   pre([next(per),id_per([_H|_T])]),
   post([wait_rp_rq(per),id_per(_T)]),
   cond([_H = idrq(_ID)])).

tr(receive_rp_rq_per,
   label([input(phy(rp_rq(_SERIE)))]),
   pre([wait_rp_rq(per),id_per(_L)]),
   post([next(per),id_per(_M)]),
   cond([concat(_SERIE,_L,_M)])).



{APERIODIQUE.........}


tr(send_id_rq_aper,
   label([output(phy(id_rq(_ID)))]),
   pre([next(aper),f_idrq([_ID|_T]),id_aper([]),aper_count(_N)]),
   post([wait_rp_rq(aper),f_idrq(_T),id_aper([]),aper_count(_M)]),
   cond([update_aper_count(_N,_M)])).


tr(receive_rp_rq_aper,
   label([input(phy(rp_rq(_SERIE)))]),
   pre([wait_rp_rq(aper),id_aper(_L)]),
   post([next(aper),id_aper(_M)]),
   cond([get_serie(_SERIE),concat(_L,_SERIE,_M)])).


tr(send_id_dat_aper,
   label([output(phy(id_dat(_ID)))]),
   pre([next(aper),id_aper([_ID|_T]),aper_count(_N)]),
   post([wait_rp_dat(_ID,aper),id_aper(_T),aper_count(_M)]),
   cond([update_aper_count(_N,_M)])).



(* ---------------- *)
(* Arbitre.cond.pro *)
(* ---------------- *)


id_per_fix([other,spec,idrq(spec)]).

update_idrq(_L,_M,_ID):- type_aper(_ID),member(_ID,_L),_M=_L,!.
update_idrq(_L,_M,_ID):- type_aper(_ID),\+ member(_ID,_L),concat(_L,[_ID],_M).
```

Annexes C : Spécifications des services périodiques et apériodiques spécifiés de Fip en L.O.T.O.S.

```
(* ============================================================ *)
(*                                                              *)
(*  SERVICE PERIODIQUE DE LA COUCHE LIAISON DE DONNEES DE FIP   *)
(*                                                              *)
(*  Modelisation du service periodique en adoptant la meme decoupe  *)
(*  que celle de PIPN (c'est a dire les etats).                 *)
(* ============================================================ *)


specification Couche_Liaison_de_donnees
              [applic_s,applic_r,med,fromedr,tomedr,fromeds,tomeds]:noexit



library Boolean endlib


(* --------------------------------- *)
(*  Abstract Data Type Specification *)
(* --------------------------------- *)



type Identificateur is Boolean   sorts Ident
 opns
  prod :-> Ident
  cons :-> Ident
  _eq_ : Ident,Ident-> Bool
  nil :-> Ident
 eqns
  forall x:Ident
   ofsort Bool
    x eq x = true;
    prod eq cons = false;
    cons eq prod = false;
endtype




type Datas is
 sorts Data
 opns
   datp :-> Data
   dat_p :-> Data
   dat_c :-> Data
endtype



type Nom_buf is
  sorts Nom_buffer
  opns
   b_dat_prod :-> Nom_buffer
   b_dat_cons :-> Nom_buffer
endtype



type Buffers is Identificateur,Datas,Nom_buf
 sorts Buffer
 opns
  create : Ident,Data,Nom_buffer -> Buffer
  get_id : Buffer -> Ident
  get_val: Buffer -> Data
  put_val : Ident,Buffer,Data->Buffer
 eqns
  forall x:Ident,y,v:Data,z:Nom_buffer,w:buffer
  ofsort Ident
   get_id(create(x,y,z)) = x;
   get_id(put_val(x,w,v))= x ;
  ofsort Data
   get_val(create(x,y,z)) = y;
   get_val(put_val(x,w,v))= v;
endtype



type Etats is Boolean
  sorts Etat
  opns
   Per :-> Etat
   Next_per,Wait_rpdat,Idle,Send,Read,Put,Get:-> Etat
  _eq_ :Etat,Etat-> Bool
  eqns
  forall x: Etat
   ofsort Bool
    x eq x =true;
    Next_per eq Wait_rpdat = false;
    Next_per eq Idle =false;
    Next_per eq Send = false;
    Next_per eq Read = false;
    Next_per eq Put = false;
    Next_per eq Get = false;
    Wait_rpdat eq Next_per = false;
    Wait_rpdat eq Idle =false;
    Wait_rpdat eq Send = false;
    Wait_rpdat eq Read = false;
    wait_rpdat eq Put = false;
    Wait_rpdat eq Get = false;
    Idle eq Next_per = false;
    Idle eq Wait_rpdat = false;
    Idle eq Send = false;
    Idle eq Read = false ;
    Idle eq Put = false ;
    Idle eq Get = false ;
    Send eq Next_per = false;
    Send eq Wait_rpdat = false;
    Send eq Read = false ;
    Send eq Put = false ;
    Send eq Get = false ;
    Send eq Idle = false;
    Read eq Next_per = false;
    Read eq Wait_rpdat = false;
    Read eq Send = false ;
    Read eq Put = false ;
    Read eq Get = false ;
    Read eq Idle = false;
    Put eq Next_per = false;
    Put eq Wait_rpdat = false;
    Put eq Send = false ;
    Put eq Read = false ;
    Put eq Get = false ;
```

```
    Put eq Idle = false;
    Get eq Next_per = false;
    Get eq Wait_rpdat = false;
    Get eq Send = false ;
    Get eq Read = false ;
    Get eq Put = false ;
    Get eq Idle = false;
endtype



type Names is Boolean,Identificateur,Etats
 sorts Name
 opns
  _eq_      : Name,Name -> Bool
  repondeur :-> Name
  station   :-> Name
  producteur,consommateur:Name,Ident -> Bool
  idstatus : Name,Ident,Etat -> Bool
 eqns
  forall x:Name
   ofsort Bool
   x eq x = true;
   repondeur eq station = false;
   station eq repondeur = false;
   consommateur(repondeur,prod) = true;
   consommateur(station,cons) = true ;
   producteur(station,prod) = true;
   producteur(repondeur,cons) = true;
   idstatus(repondeur,cons,per) = true;
   idstatus(station,prod,per) = true;
endtype



type Primitives is
  sorts Primitive
  opns
    Iddat, Rpdat, L_get_dem, L_get_conf, L_put_dem, L_put_conf, L_send_indic,
    L_received_indic :-> Primitive
endtype



type Liste is Boolean,Identificateur
sorts Liste
  opns
   creerliste :-> Liste
   add: Ident, Liste -> Liste
   first: Liste -> Ident
   tail: Liste,Ident-> Liste
   liste_vide: Liste-> Bool
  eqns
   forall x,v:Ident, y,z:Liste
    ofsort Ident
     first(add(x,creerliste)) = x ;
     first(add(x,y)) = first(y);
    ofsort Bool
     liste_vide(creerliste) = true;
     liste_vide(add(x,y)) = false;
    ofsort Liste
     tail(add(x,add(v,creerliste)),v) = add(x,creerliste);
     tail(add(x,y),x) =y  ;
endtype


(* ----------------------- *)
(* Behaviour specification *)
(*------------------------ *)

behaviour
  hide med,medr,meds in
  (Arbitre[med] (creerliste,Next_per)
    |||
   Repondeur[fromedr,tomedr,applic_r] (Idle,repondeur,
                                       create(cons,dat_c,b_dat_prod),
                                       create(prod,dat_p,b_dat_cons),Nil)
    |||
   Station[fromeds,tomeds,applic_s] (Idle,station,
                                     create(prod,dat_p,b_dat_prod),
                                     create(cons,dat_c,b_dat_cons),Nil)
  )

|[med,fromedr,tomedr,fromeds,tomeds]|

   Medium[med,fromedr,tomedr,fromeds,tomeds] (Idle)



where


 (* - - - - - - - - - - *)
 (* PROCESSUS Arbitre *)
 (* - - - - - - - - - - *)

 process Arbitre[med](l:Liste,s:Etat):noexit :=

 [(liste_vide(l)) and (s eq Next_per)]
    -> i;
       Arbitre[med] (add(cons,add(prod,l)),Next_per)

[]

 [(not(liste_vide(l))) and  (s eq Next_per)]
    -> med !Iddat !first(l);
       Arbitre[med] (tail(l,first(l)),Wait_rpdat)

 []
 [s eq Wait_rpdat]
    -> med !Rpdat ?val:Data;
       Arbitre[med] (l,Next_per)

 endproc



 (* - - - - - - - - - - *)
 (* PROCESSUS Medium  *)
 (* - - - - - - - - - - *)


process Medium[med,fromedr,tomedr,fromeds,tomeds] (s:Etat):noexit :=

 [s eq Idle] ->
        med ?prim:primitive  ?param_id:Ident;
```

```
            (fromeds !prim !param_id ;
             fromedr !prim !param_id ;
            ( Medium[med,fromedr,tomedr,fromeds,tomeds](Idle)))
        []
        tomedr ?prim:primitive  ?param_val:Data ;
              (med !prim !param_val;
               fromeds!prim !param_val;
              (Medium[med,fromedr,tomedr,fromeds,tomeds](Idle)))
        []
        tomeds ?prim:primitive  ?param_val:Data ;
              (med !prim !param_val;
               fromedr!prim !param_val;
              (Medium[med,fromedr,tomedr,fromeds,tomeds] (Idle)))

endproc



 (* - - - - - - - - - - *)
 (* PROCESSUS Repondeur *)
 (* - - - - - - - - - - *)



process Repondeur[fromed,tomed,applic]
        (s:Etat,nom:Name,b_dat_p:Buffer,b_dat_c:Buffer,id:Ident):noexit:=


[(s eq Idle) and (nom eq repondeur)]
    -> (fromed !Iddat ?id:Ident[producteur(nom,id)];
        Repondeur[fromed,tomed,applic] (Send,nom,b_dat_p,b_dat_c,id)

        []

        fromed !Iddat ?id:ident [consommateur(nom,id)];
        Repondeur[fromed,tomed,applic] (Read,nom,b_dat_p,b_dat_c,id)
       )

 []

 [s eq Send]
    -> tomed !Rpdat !get_val(b_dat_p)  [idstatus(nom,id,per)];
       (applic !L_send_indic !id !get_val(b_dat_p);
        Repondeur[fromed,tomed,applic](Idle,nom,b_dat_p,b_dat_c,id))

 []

 [s eq Read]
     -> fromed !Rpdat ?val1:Data;
        (applic !L_received_indic !id;
        Repondeur[fromed,tomed,applic]
                 (Idle,nom,b_dat_p,put_val(get_id(b_dat_c),b_dat_c,val1),id))

endproc



 (* - - - - - - - - - - *)
 (* PROCESSUS Station *)
 (* - - - - - - - - - - *)
```

```
process Station[fromed,tomed,applic]
          (s:Etat,nom:Name,b_dat_p:Buffer,b_dat_c:Buffer,id:Ident):noexit:=


[s eq Idle]
   ->(
     (applic !L_put_dem ?id:Ident ?val1:Data [producteur(nom,id)];
      (Station[fromed,tomed,applic]
               (Put,nom,put_val(id,b_dat_p,val1),b_dat_c,id)))

    []

     (applic !L_get_dem ?id:Ident [consommateur(nom,id)];
      (Station[fromed,tomed,applic] (Get,nom,b_dat_p,b_dat_c,id)))

    []

     ([nom eq station]
       -> (fromed !Iddat ?id:Ident [producteur(nom,id)];
           (Station[fromed,tomed,applic] (Send,nom,b_dat_p,b_dat_c,id)))

          []

          (fromed !Iddat ?id:Ident [consommateur(nom,id)];
           (Station[fromed,tomed,applic] (Read,nom,b_dat_p,b_dat_c,id)))
     )
     )

[]

[s eq Send]
   -> tomed !rpdat !get_val(b_dat_p) [idstatus(nom,id,per)];
      (applic !L_send_indic !id !get_val(b_dat_p);
      (Station[fromed,tomed,applic] (Idle,nom,b_dat_p,b_dat_c,id)))

[]

[s eq Read]
    -> fromed !rpdat ?val1:Data;
       (applic !L_received_indic !id;
        Station[fromed,tomed,applic]
               (Idle,nom,b_dat_p,put_val(get_id(b_dat_c),b_dat_c,val1),id))

[]

[s eq Put]
   -> applic !l_put_conf !id;
      Station[fromed,tomed,applic] (Idle,nom,b_dat_p,b_dat_c,id)

[]

[s eq Get]
   -> applic !L_get_conf !id !get_val(b_dat_c);
      Station[fromed,tomed,applic] (Idle,nom,b_dat_p,b_dat_c,id)


endproc
endspec
```

```
(* ================================================================== *)
(*                                                                    *)
(*  SERVICE PERIODIQUE DE LA COUCHE LIAISON DE DONNEES DE FIP         *)
(*                                                                    *)
(*  Transformation des reseaux predicats/transitions en Lotos sur base    *)
(*  de la synchronisation des messages.                               *)
(*  La notion d'etats disparait completement                          *)
(* ================================================================== *)


Specification Couche_Liaison_de_donnees
            [applic_s,applic_r,med,fromedr,tomedr,fromeds,tomeds]:noexit



(*library Boolean endlib*)

type     Boolean is
sorts    Bool
opns     true, false                    : -> Bool
         not                            : Bool -> Bool
         _and_, _or_, _xor_, _implies_, _iff_, _eq_, _ne_
                                        : Bool, Bool -> Bool

eqns     forall x, y : Bool

         ofsort Bool
         not(true)      = false   ;
         not(false)     = true    ;

         x and true     = x       ;
         x and false    = false   ;

         x or true      = true    ;
         x or false     = x       ;

         x xor y        = (x and not(y)) or (y and not(x)) ;
         x implies y    = y or not(x)                      ;
         x iff y        = (x implies y) and (y implies x)  ;
         x eq y         = x iff y ;
         x ne y         = x xor y ;
endtype



(* ---------------------------------- *)
(*  Abstract Data Types Specifications  *)
(* ---------------------------------- *)


type Identificateur is Boolean
 sorts Ident
 opns
  prod :-> Ident
  cons :-> Ident
  _eq_ : Ident,Ident-> Bool
 eqns
  forall x:Ident
   ofsort Bool
    x eq x = true;
    prod eq cons = false;
    cons eq prod = false;
endtype


type Datas is
 sorts Data
 opns
   datp :-> Data
   dat_p :-> Data
   dat_c :-> Data
endtype


type Nom_buf is
  sorts Nom_buffer
  opns
   b_dat_prod :-> Nom_buffer
   b_dat_cons :-> Nom_buffer
endtype



type Buffers is Identificateur,Datas,Nom_buf
 sorts Buffer
 opns
  create : Ident,Data,Nom_buffer -> Buffer
  get_id : Buffer -> Ident
  get_val: Buffer -> Data
  put_val : Ident,Buffer,Data->Buffer
 eqns
  forall x:Ident,y,v:Data,z:Nom_buffer,w:buffer
  ofsort Ident
   get_id(create(x,y,z)) = x;
   get_id(put_val(x,w,v))= x ;
  ofsort Data
   get_val(create(x,y,z)) = y;
   get_val(put_val(x,w,v))= v;
endtype


 type Etats is Boolean
  sorts Etat
  opns
   per :-> Etat
 endtype



type Names is Boolean,Identificateur,Etats
 sorts Name
 opns
  _eq_      : Name,Name -> Bool
  repondeur :-> Name
  station   :-> Name
  producteur,consommateur:Name,Ident -> Bool
  idstatus : Name,Ident,Etat -> Bool
 eqns
  forall x:Name
   ofsort Bool
   x eq x = true;
   repondeur eq station = false;
   station eq repondeur = false;
   consommateur(repondeur,prod) = true;
   consommateur(station,cons) = true ;
   producteur(station,prod) = true;
```

```
   producteur(repondeur,cons) = true;
   idstatus(repondeur,cons,per) = true;
   idstatus(station,prod,per) = true;
endtype



type Primitives is
  sorts Primitive
  opns
    Iddat, Rpdat, L_get_dem, L_get_conf, L_put_dem, L_put_conf, L_send_indic,
    L_received_indic :-> Primitive
endtype



type Liste is Boolean,Identificateur
sorts Liste
  opns
   creerliste :-> Liste
   add: Ident, Liste -> Liste
   first: Liste -> Ident
   tail: Liste,Ident-> Liste
   liste_vide: Liste-> Bool
  eqns
   forall x,v:Ident, y,z:Liste
    ofsort Ident
     first(add(x,creerliste)) = x ;
     first(add(x,y)) = first(y);
    ofsort Bool
     liste_vide(creerliste) = true;
     liste_vide(add(x,y)) = false;
    ofsort Liste
     tail(add(x,add(v,creerliste)),v) = add(x,creerliste);
     tail(add(x,y),x) =y  ;
endtype



(* ------------------------------------ *)
(*  Behaviour Specification             *)
(* ------------------------------------ *)



behaviour
  hide med,fromedr,tomedr,tomeds,fromeds in
  (Arbitre[med] (add(cons,add(prod,creerliste)))
    |||
   Repondeur[fromedr,tomedr,applic_r]
    (repondeur,create(cons,dat_c,b_dat_prod),create(prod,dat_p,b_dat_cons))
    |||
   Station[fromeds,tomeds,applic_s]
    (station,create(prod,dat_p,b_dat_prod),create(cons,dat_c,b_dat_cons)))
|[med,fromedr,tomedr,fromeds,tomeds]|

   Medium[med,fromedr,tomedr,fromeds,tomeds]



where
```

```
(* ------------------------------------ *)
(* PROCESS Arbitre                      *)
(* ------------------------------------ *)


process Arbitre[med](l:Liste):noexit:=

 [(liste_vide(l))] -> stop
 []
 [(not(liste_vide(l)))] -> med !Iddat !first(l);
                           (med !Rpdat ?val:Data;
                          Arbitre[med] (tail(l,first(l))))

 endproc



(* ------------------------------------ *)
(*  PROCESS Medium                      *)
(* ------------------------------------ *)

process Medium[med,fromedr,tomedr,fromeds,tomeds]:noexit :=

        med ?prim:primitive  ?param_id:Ident;
            (fromeds !prim !param_id ;
             fromedr !prim !param_id ;
            ( Medium[med,fromedr,tomedr,fromeds,tomeds]))
        []
        tomedr ?prim:primitive  ?param_val:Data ;
              (med !prim !param_val;
               fromeds!prim !param_val;
              (Medium[med,fromedr,tomedr,fromeds,tomeds]))
        []
        tomeds ?prim:primitive  ?param_val:Data ;
              (med !prim !param_val;
               fromedr!prim !param_val;
              (Medium[med,fromedr,tomedr,fromeds,tomeds]))

endproc



(* ------------------------------------ *)
(*  PROCESS Repondeur                   *)
(* ------------------------------------ *)


process Repondeur[fromed,tomed,applic]
                (nom:Name,b_dat_p:Buffer,b_dat_c:Buffer):noexit:=


 (fromed !Iddat ?id:Ident [producteur(nom,id)];
  tomed !Rpdat !get_val(b_dat_p) [idstatus(nom,get_id(b_dat_p),per)];
         (applic !L_send_indic !id  !get_val(b_dat_p);
                 Repondeur[fromed,tomed,applic](nom,b_dat_p,b_dat_c)))


 []

 (fromed !Iddat ?id:ident [consommateur(nom,id)];
  fromed !Rpdat ?val1:Data;
```

```
        (applic !L_received_indic !id;
        Repondeur[fromed,tomed,applic](nom,b_dat_p,put_val(id,b_dat_c,val1))))

endproc



(* ------------------------------------ *)
(*  PROCESS Station                     *)
(* ------------------------------------ *)


process Station[fromed,tomed,applic]
              (nom:Name,b_dat_p:buffer,b_dat_c:buffer):noexit:=



(applic !L_put_dem ?id:Ident ?val1:Data [producteur(nom,id)];
       (applic !L_put_conf !id;
       Station[fromed,tomed,applic](nom,put_val(id,b_dat_p,val1),b_dat_c)))
[]

(applic !L_get_dem ?id:Ident [consommateur(nom,id)];
         (applic !L_get_conf !id !get_val(b_dat_c);
         Station[fromed,tomed,applic](nom,b_dat_p,b_dat_c)))

[]


(fromed !Iddat ?id:Ident [producteur(nom,id)];
        (tomed !Rpdat !get_val(b_dat_p) [idstatus(nom,id,per)];
         applic !L_send_indic !id !get_val(b_dat_p);
         Station[fromed,tomed,applic](nom,b_dat_p,b_dat_c)))

[]

(fromed !Iddat ?id:Ident [consommateur(nom,id)];
        (fromed !Rpdat ?val1:Data;
         applic !L_received_indic !id ;
         Station[fromed,tomed,applic](nom,b_dat_p,put_val(id,b_dat_c,val1))))


endproc
endspec
```

```
(* ========================================================================= *)
(* SERVICE APERIODIQUE SPECIFIE DE LA COUCHE LIAISON DE DONNEES              *)
(*                                                                           *)
(* Modelisation du service aperiodique specifie de la couche liaison sur     *)
(* de la synchronisation des messages.                                       *)
(* ========================================================================= *)


Specification Couche_Liaison_de_donnees
              [applic_s,applic_r,med,fromedr,tomedr,fromeds,tomeds]:noexit



library Boolean endlib


(* ----------------------------------- *)
(*  Abstract Data Types Specifications  *)
(* ----------------------------------- *)


type Identificateur is Boolean
 sorts Ident
 opns
  other :-> Ident
  spec :-> Ident
  _eq_ : Ident,Ident-> Bool
 eqns
  forall x:Ident
   ofsort Bool
    x eq x = true;
    spec eq other = false;
    other eq spec = false;

endtype


type Datas is
 sorts Data
 opns
   datp :-> Data
   dat_p :-> Data
   dat_c :-> Data
   data :-> Data
endtype


type Nom_buf is
  sorts Nom_buffer
  opns
   b_dat_prod :-> Nom_buffer
   b_dat_cons :-> Nom_buffer
   b_req :-> Nom_buffer
endtype


type Compteurs is Boolean
   sorts Compteur
   opns
    0 :-> Compteur
    1 :-> Compteur
    succ:Compteur->Compteur
    _eq_:Compteur,Compteur->Bool
   eqns
    forall c,o :Compteur
    ofsort Compteur
    succ(0) = 1;
    ofsort Bool
    0 eq 0 = true;
    0 eq 1 = false;
    1 eq 0 = false;
    1 eq 1 = true;
    succ(c)  eq succ(c) = true;
    c eq succ(c) = false;
    succ(c) eq c = false;

 endtype


type Buffers is Identificateur,Datas,Nom_buf
 sorts Buffer
 opns
  create : Ident,Data,Nom_buffer -> Buffer
  get_id : Buffer -> Ident
  get_val: Buffer -> Data
  put_val : Ident,Buffer,Data->Buffer
 eqns
  forall x:Ident,y,v:Data,z:Nom_buffer,w:buffer
  ofsort Ident
   get_id(create(x,y,z)) = x;
   get_id(put_val(x,w,v))= x ;
  ofsort Data
   get_val(create(x,y,z)) = y;
   get_val(put_val(x,w,v))= v;
endtype


 type Etats is Boolean
  sorts Etat
  opns
   per ,aper_spec_req:-> Etat
 endtype



type Names is Boolean,Identificateur,Etats,Compteurs
 sorts Name
 opns
  _eq_       : Name,Name -> Bool
  repondeur :-> Name
  station   :-> Name
  producteur,consommateur:Name,Ident -> Bool
  idstatus : Name,Ident,Etat -> Bool
  no_rq_spec_cond : Compteur,Name,Ident -> Bool
 eqns
  forall x:Name
   ofsort Bool
   x eq x = true;
   repondeur eq station = false;
   station eq repondeur = false;
   consommateur(repondeur,spec) = true;
   consommateur(station,other) = true ;
   producteur(station,spec) = true;
   producteur(repondeur,other) = true;
```

```
   idstatus(repondeur,other,per) = true;
   idstatus(station,spec,aper_spec_req) = true;
   no_rq_spec_cond(0,station,spec) = true;
   no_rq_spec_cond(0,station,other) = true;
endtype



type Primitives is
  sorts Primitive
  opns
    Iddat, Rpdat, L_get_dem, L_get_conf, L_put_dem, L_put_conf, L_send_indic,
    L_received_indic,Idrq,Rprq,Rpdat_rq,ecrase,
    L_update_spec_dem,L_update_spec_conf :-> Primitive
endtype



type Liste is Boolean,Identificateur
sorts Liste
  opns
   creerliste :-> Liste
   add: Ident, Liste -> Liste
   concat : Liste,Liste -> Liste
   first: Liste -> Ident
   tail: Liste,Ident-> Liste
   liste_vide: Liste-> Bool
   membre : Ident,Liste -> Bool
   serie : Liste -> Bool
   eqns
   forall x,v:Ident, y,z:Liste
    ofsort Ident
     first(add(x,creerliste)) = x ;
     first(add(x,y)) = first(y);
     first(concat(y,z)) = first(y);
    ofsort Bool
     liste_vide(creerliste) = true;
     liste_vide(add(x,y)) = false;
     liste_vide(concat(z,y))= liste_vide(z) and liste_vide(y);
     membre(x,creerliste) = false;
     membre(x,add(x,creerliste)) = true;
     serie(add(other,creerliste)) = true;
     ofsort Liste
     tail(add(x,add(v,creerliste)),v)=add(x,creerliste);
     tail(add(x,y),x) =y ;

     concat(add(x,creerliste),y) = add(x,y);

endtype


type Buffer_requete is Identificateur,Liste,Nom_buf,Compteurs
 sorts buf_req
 opns
  create : Nom_buffer,Ident,Liste,Compteur-> buf_req
  get_rq : buf_req -> Compteur
  get_serie:buf_req -> Liste
  inchange:buf_req -> buf_req
  put_rq_1 : Nom_buffer,Ident,Liste,Compteur-> buf_req
  reinit_rq :Nom_buffer,Ident,Liste,Compteur -> buf_req
   eqns
  forall x:Ident,y,v:Liste,z:Nom_buffer,w:buf_req,c :Compteur
  ofsort Compteur
  get_rq(create(z,x,y,c)) = c;
  get_rq(put_rq_1(z,x,y,c)) = c;
  get_rq(reinit_rq(z,x,y,c)) = c;
  ofsort Liste
  get_serie(create(z,x,y,c)) = y;
  get_serie(put_rq_1(z,x,y,c)) = y;
  get_serie(reinit_rq(z,x,y,c))=y;
  ofsort buf_req
  inchange(create(z,x,y,c)) = create(z,x,y,c);
  inchange(put_rq_1(z,x,y,c)) = put_rq_1(z,x,y,c);
  inchange(reinit_rq(z,x,y,c)) = reinit_rq(z,x,y,c);


 endtype


(* ------------------------------------ *)
(*  Behaviour Specification             *)
(* ------------------------------------ *)



behaviour
 hide med,fromedr,tomedr,tomeds,fromeds in
 (Arbitre[med]
        (add(spec,add(other,creerliste)),creerliste,creerliste,0)
    |||
   Repondeur[fromedr,tomedr,applic_r]
    (repondeur,create(other,data,b_dat_prod),create(spec,data,b_dat_cons))
    |||
   Station[fromeds,tomeds,applic_s]
    (station,create(spec,data,b_dat_prod),create(other,data,b_dat_cons),
     create(b_req,spec,creerliste,0))
  )

|[med,fromedr,tomedr,fromeds,tomeds]|

   Medium[med,fromedr,tomedr,fromeds,tomeds]



where


(* ------------------------------------ *)
(* PROCESS Arbitre                      *)
(* ------------------------------------ *)


process Arbitre[med]
        (l_per:Liste,F_idrq:Liste,l_aper:Liste,apercount:Compteur):noexit:=


([liste_vide(l_per) and (not(liste_vide(F_idrq))) and (liste_vide(l_aper))]
    -> med !Idrq !first(F_idrq);
       med !Rprq ?serie:Liste ;
       arbitre[med]
       (l_per,tail(F_idrq,first(F_idrq)),
              concat(serie,l_aper),succ(apercount))
)
```

```
[]


([(not(liste_vide(l_per)))]
    -> med !Iddat !first(l_per);
       (med !Rpdat ?val:Data;
       Arbitre[med] (tail(l_per,first(l_per)),F_idrq,l_aper,apercount)

       []

       ([(first(l_per) eq spec) and (membre(first(l_per),F_idrq))]
        ->  med !Rpdat_rq ?val:data ;
            Arbitre[med] (tail(l_per,first(l_per)),F_idrq,l_aper,apercount))

       []

       ([(first(l_per) eq spec) and (not(membre(first(l_per),F_idrq)))]
        -> med !Rpdat_rq ?val:data ;
           Arbitre[med] (tail(l_per,first(l_per)),add(first(l_per),F_idrq),
                         l_aper,apercount))
       )
)


[]


([(liste_vide(l_per) and (not(liste_vide(l_aper))))]
   -> med !Iddat !first(l_aper);
      med !Rpdat ?val:Data;
      arbitre[med](l_per,F_idrq,tail(l_aper,first(l_aper)),succ(apercount))
)


[]


([(liste_vide(l_per) and (liste_vide(F_idrq) and (liste_vide(l_aper))))]
   -> i;
      Arbitre[med](add(spec,add(other,creerliste)),creerliste,creerliste,0)

)


endproc



(* ------------------------------------ *)
(*  PROCESS Medium                      *)
(* ------------------------------------ *)

process Medium[med,fromedr,tomedr,fromeds,tomeds]:noexit :=


med ?prim:primitive  ?param_id:Ident;

(fromeds !prim !param_id ;
 fromedr !prim !param_id ;
 ( Medium[med,fromedr,tomedr,fromeds,tomeds]))


[]

tomedr ?prim:primitive  ?param_val:Data ;
 (med !prim !param_val;
  fromeds!prim !param_val;
  (Medium[med,fromedr,tomedr,fromeds,tomeds]))


[]

tomeds ?prim:primitive  ?param_val:Data ;
 (med !prim !param_val;
  fromedr !prim !param_val;
  (Medium[med,fromedr,tomedr,fromeds,tomeds]))

[]

tomeds ?prim:primitive  ?list:Liste;
 (med !prim !list;
  fromedr !prim !list;
  (Medium[med,fromedr,tomedr,fromeds,tomeds]))


endproc



(* ------------------------------------ *)
(*  PROCESS Repondeur                   *)
(* ------------------------------------ *)


process Repondeur[fromed,tomed,applic]
               (nom:Name,b_dat_p:Buffer,b_dat_c:buffer):noexit:=



(fromed !Iddat ?id:Ident [producteur(nom,id)];
 tomed !Rpdat !get_val(b_dat_p) [idstatus(nom,get_id(b_dat_p),per)];
 applic !L_send_indic !id  !get_val(b_dat_p);
 Repondeur[fromed,tomed,applic](nom,b_dat_p,b_dat_c)
)


[]

(fromed !Iddat ?id:ident [consommateur(nom,id)];
 (fromed !Rpdat ?val1:Data;
  applic !L_received_indic !id;
  Repondeur[fromed,tomed,applic] (nom,b_dat_p,put_val(id,b_dat_c,val1))
 )

 []

 (fromed !Rpdat_rq  ?val1:Data;
  applic !L_received_indic !id;
  Repondeur[fromed,tomed,applic] (nom,b_dat_p,put_val(id,b_dat_c,val1))
 )

)


[]
```

```
(fromed !Idrq ?id:ident;
 fromed !Rprq ?serie:liste;
 Repondeur[fromed,tomed,applic] (nom,b_dat_p,b_dat_c)
)

endproc



(* ------------------------------------ *)
(*  PROCESS Station                     *)
(* ------------------------------------ *)


process Station[fromed,tomed,applic]
               (nom:Name,b_dat_p:buffer,b_dat_c:buffer,b_req:buf_req):noexit:=



([get_rq(b_req) eq 0]
   ->  applic !L_update_spec_dem ?id:Ident ?serie:Liste [serie(serie)];
       Station[fromed,tomed,applic]
               (nom,b_dat_p,b_dat_c,put_rq_1(b_req,id,serie,succ(0)))
)

[]


([get_rq(b_req) eq succ(0)]
   ->  applic !L_update_spec_dem ?id:Ident ?serie:Liste [serie(serie)];
       applic !L_update_spec_conf !id !ecrase;
       Station[fromed,tomed,applic]
               (nom,b_dat_p,b_dat_c,put_rq_1(b_req,id,serie,succ(0)))
)


[]


(fromed !Iddat ?id:Ident [consommateur(nom,id)];
 fromed !Rpdat ?val:Data;
 applic !L_received_indic !id ;
 Station[fromed,tomed,applic] (nom,b_dat_p,b_dat_c,inchange(b_req))
)


[]


(fromed !Idrq ?id:Ident;
 ([get_rq(b_req) eq 0]
    -> tomed !Rprq  !creerliste;
       Station[fromed,tomed,applic] (nom,b_dat_p,b_dat_c,inchange(b_req))
  []

 ([get_rq(b_req) eq 1]
    -> tomed !Rprq  !get_serie(b_req) ;
       Station[fromed,tomed,applic]
               (nom,b_dat_p,b_dat_c,reinit_rq(b_req,id,get_serie(b_req),0))
 )
 )
)


[]

(fromed !Iddat ?id:Ident [producteur(nom,id)];
  (tomed !Rpdat !get_val(b_dat_p) [idstatus(nom,id,per)];
   applic !L_send_indic !id !get_val(b_dat_p);
   Station[fromed,tomed,applic] (nom,b_dat_p,b_dat_c,inchange(b_req))


  []

   ([get_rq(b_req) eq 0]
      -> tomed !Rpdat !get_val(b_dat_p);
         applic !L_send_indic !id !get_val(b_dat_p);
         Station[fromed,tomed,applic] (nom,b_dat_p,b_dat_c,inchange(b_req))
   )


  []



  ([get_rq(b_req) eq 1]
     -> tomed !Rpdat_rq !get_val(b_dat_p) [idstatus(nom,id,aper_spec_req)];
        applic !L_send_indic !id !get_val(b_dat_p);
        Station[fromed,tomed,applic](nom,b_dat_p,b_dat_c,inchange(b_req))
  )

  )



)


endproc
endspec
```

Bibliographie

[Az-Ve-LL 89] P. AZEMA, F. VERNADAT, J.C. LLORET, "Spécification logique de protocoles de communication à l'aide des réseaux prédicat/transition étiquettés", LAAS report n°89015, Toulouse, Janvier 1989

[Bo-Su 80] BOCHMANN, SUSHINE, "Formal Methods in Communication Protocol", IEEE transactions on communication, April 1980

[Courtiat 87] J.P. COURTIAT, "Contribution à la description formelle de protocoles", thèse présentée à l'université Paul Sabatier de Toulouse, 1987

[ISO 87] ISO/TC 97, "Criteria for the Use and Applicability of Formal Description Techniques", New York, October 1987

[ISO 87] ISO/TC 97/SC 21, "Architectural Semantics for Formal Description Techniques", USA, June 1987

[ISO 88] ISO/IEC JTC1/SC21, "Proposed draft technical report on Guidelines for the application of Estelle, LOTOS, and SDL", 1988

[ISO 89] "Information processing systems - Open systems interconnection - LOTOS - A formal description technique based on the temporal ordering of observational behaviour", ISO 8807, 1989

[Leduc 87] G.J. LEDUC, "LOTOS, un outil utile ou un autre langage académique?", 9e journée francophone sur l'informatique, Liège, Janvier 1987

[Le-Th 89] P. LETERRIER, J.P. THOMESE, "Fonctions de base d'un bus de terrain", Minis&Micros, octobre 1989

[Lloret&al 88] J.P. LLORET, P. AZEMA, F. VERNADAT, "Réseaux PrT labellés : structuration et composition",

[LLoret 88] J.C. LLORET, "PIPN ; Environnement de prototypage et de vérification d'algorithmes répartis", Manuel de référence version 1, Toulouse, Janvier 1988

[LOTOS 89] P.H.J. VAN EIJK et al (Eds), "The Formal Description Technique LOTOS", Results of the ESPRIT/ SEDOS Project, North-Holland 1989

[Mi lner 80] R. MILNER, "C.C.S., a calculus of Communicating systems", Lecture Notes on Computer Science, 92, 1980

[Najm 86] E. Najm, "Présentation du langage de spécification LOTOS", 3e Congrès De Nouvelles Architectures pour les Communications, Paris, Octobre 1986

[Peterson 81] J.L. PETERSON, "Petri net theory and the modelling of systems", Prentice-Hall 1981

[SEDOS 88] "SEDOS : Software Environment for the Design of Open distributed Systems", final SEDOS Report , Project ST 410, May 1988

Table des matières